Resource Management for the Tagged Token Dataflow Architecture

by

David E. Culler

January, 1980

This research was supported in part by the Advanced Research Projects Agency under contract N00014-75-0661 and in part by various grants from the International Business Machines Corporation.

MASSACHUSETTS INSTITUTE of TECHNOLOGY
Laboratory for Computer Science

Cambridge Massachusetts 02139

RESOURCE MANAGEMENT FOR THE TAGGED TOKEN DATAFLOW ARCHITECTURE

by

David E. Culler

Submitted to the Department of Electrical Engineering and Computer Science
on January 18, 1985 in partial fulfillment of the requirements
for the Degree of Master of Science

Abstract
The Tagged Token Dataflow Architecture is multiprocessor based on the U-interpreter model of dataflow computation. It captures the essential execution mechanism of the U-interpreter precisely; operations are enabled for execution by the availability of operand data. However, computational resources in the model and the machine are viewed quite differently. This thesis addresses four major resource management issues essential to bridge the gap between the U-interpreter and the Tagged Token Dataflow Architecture.

1. *Termination detection*: The completion of code-block invocations must be detected so resources can be released and reused. This problem is solved by augmenting graphs with auxiliary arcs so that the execution of a particular operation signifies completion.

2. *Token store overflow*: Deadlock due to token store overflow is overcome by determining the worst-case token storage requirements of code-blocks in advance and reserving that amount of token storage. Token storage requirements are determined by modeling the space of legal graph configurations as a system of integer linear constraints and maximizing the token storage requirement subject to these constraints. The resulting integer linear program is of a form that can be solved efficiently.

3. *Iteration identifier overflow*: The labels carried on tokens are represented by fixed size tags; iteration identifiers must be reused to allow loops to execute a large number of iterations. This requires controlling the unfolding (*i.e.*, the number of concurrent iterations) of loops. It is shown that loops exhibit bounded unfolding if and only if the graph forms a single strongly connected component. A transformation is proposed which allows controlled unfolding of loops and automatic reuse of iteration identifiers.

4. *Program deadlock*: Each code-block invocation requires certain resources. Thus, the resource capacity of the machine limits the number of concurrent invocations. Programs unfold as a tree of invocations. If too much parallelism is exposed (*i.e.*, if the invocation tree is allowed to grow too broad) the machine resources will become prematurely exhausted and cause the program to deadlock. A resource management strategy is developed which constrains program unfolding so that just enough parallelism is exposed to fully utilize the machine.

Thesis Supervisor: Dr. Arvind

Title: Associate Professor of Electrical Engineering and Computer Science

Keywords: Dataflow, U-interpreter, Tagged Token Dataflow Architecture, Resource Management, Termination Detection, Token Storage, Tag Management, Controlled Loop Unfolding, Program Unfolding, Program Deadlock, Bounded Resource Programs.

Acknowledgments

There are many people I want to thank. People without whom this work would not have been possible. Sara, my wife, I thank you with all my heart. You have sacrificed so much to allow me to pursue graduate studies and offered your love, patience, and support. I can hardly express how much it all means to me. My family, as always, offered unending support and confidence. Arvind, my advisor, offered a quiet mixture of freedom, guidance, and support. Steve Heller was an absolute mench, reading many drafts and giving me a hard time in the most considerate ways. Greg Papadapoulos was invaluable in the initial formulation of many of the ideas and read the early drafts. Bhaskar GuhaRoy shared his own gift for enthusiasm. Deborah Estrin listened patiently to the trials and tribulations while we jogged along the Charles. Professor Leiserson started me thinking about constraint systems. Finally, all the other members of the dataflow community here at M.I.T have provided a cheerful and stimulating research environment. I want to thank you all.

This research was supported in part by the Advanced Research Projects Agency under contract N00014-75-0661 and in part by various grants from the International Business Machines Corporation.

A fine sunrise or an elegant sunset
achieves moving colors and masses of changing light
in a properly organized confusion.

Carl Sandburg

Dedicated to Sara

with all my love,

D.

Table of Contents

Chapter One

Introduction

The Tagged Token Dataflow Architecture is a novel multiprocessor architecture based on the U-interpreter [9] model of dataflow computation. The machine is extremely close to the model in certain aspects, but fundamentally different in others. In both the model and the machine, operations are enabled for execution based on the availability of data. However, the model assumes unbounded resources and unbounded processing power, whereas the machine must operate within finite resource constraints. This difference raises a variety of resource management problems which must be overcome for large programs to execute effectively on the Tagged Token Dataflow Architecture

The execution mechanism embodied in the U-interpreter and realized in the Tagged Token Dataflow Architecture is quite unlike that of conventional computers. Under the U-interpreter, a program is a *dataflow* graph; the nodes denote operations, and the arcs denote data dependence between operations. Data values are carried on labeled *tokens*, which flow along the arcs in the graph. An operation may execute (or *fire*) whenever a set of tokens is available on its input arcs. When an operation fires it consumes a token from each input arc and produces a result token on each output arc. The Tagged Token Dataflow Architecture captures these aspects of the model precisely. A program for this machine is an adjacency list representation of a dataflow graph; each instruction contains an operation code and a list of successor instruction addresses. Data is transferred between instructions as labeled packets of information, *i.e.*, tokens. Instructions are scheduled based soley on the availability of operands; an associative token store detects when a matched set of operands is available and enables the corresponding instruction for execution. When an instruction executes, its operands are purged from the token store, a result is computed, and result tokens are generated with labels corresponding to the output arcs of the instruction.

The fundamental difference between the model and the machine is the viewpoint adopted toward computational resources. The model is essentially free of resource constraints. All resources are unbounded, allocated implicitly, and used only once. No bounds are placed on the number of tokens which may reside on a arc, the number of simultaneously executing operations, or the size of the labels carried on tokens. A program is allowed to unfold in an unconstrained manner, exposing as much parallel activity as possible [7]. In the Tagged Token Dataflow Architecture, resources are in finite supply, allocated explicitly, and reused by necessity. A given machine configuration offers a fixed amount of token storage and a fixed number of processors. The labels carried on tokens are represented by fixed size tags. The viewpoint adopted in the U-interpreter model is simple and elegant, but impractical for a realistic machine.

This thesis offers a way to overcome the differences between the model and the machine through a concerted approach to resource management, involving the compiler and the run-time system. Program graphs based on the U-interpreter model are transformed into equivalent graphs which are more suitable for execution on the Tagged Token Dataflow Architecture. These graphs have predictable resource requirements and include special operations to engage the run-time system. The run-time system has two responsibilities: dynamic allocation/deallocation of resources, and dynamic control of program execution. The work presented here is motivated by the need to resolve particular resource management problems facing the Tagged Token Dataflow Architecture, however, it serves a more general goal as well. Resource management is a fundamental aspect of any dataflow machine, and the issues raised in this thesis should have a prominent role in the design and evaluation of dataflow architectures in general.

1.1. Overview of the Thesis

The U-interpreter

The U-interpreter is presented in Chapter 2. Under this model, tokens carry ***activity names***, in addition to data values, which specify the part of the computation to which the token belongs. An operation may execute whenever a set of tokens with identical activity names is available on its input arcs. Parallelism and synchronization are expressed naturally through the branching and rejoining of the graph. A program graph establishes a partial order on the firing of operations; if there is no dependency between two operations, they are permitted to fire in parallel or in any order. The basic operations are described by simple rules for producing result values and activity names from input values and activity names. The operations have no side-effects and no internal state. Thus, if there is no dependency between two operations, the results produced are not affected by the relative order in which the operations fire. The class of legal program graphs is defined by a set of graph schemata and simple composition rules. These graphs are structured such that all legal execution orderings produce exactly the same results. The model allows unbounded queuing of tokens on the arcs, unbounded activity names, and unbounded parallel activity.

In examining the U-interpreter, there are a number of important points to observe. (1) A program unfolds as a tree of *code-block invocations*, much as a conventional program does, except that many branches of the tree may be in execution simultaneously. At any point in a computation, the active portion of the invocation tree represents the *state* of the computation. (2) Activity names grow rapidly in size. The *context* portion of the activity name specifies a path from the root of the tree to a particular code-block invocation, and thus grows in size linearly with the depth of the invocation tree. The *iteration* number portion of the activity name grows logarithmically with the number of iterations executed by a loop. (3) The token storage requirement of a program is the maximum number of tokens that co-exist at any point in a legal execution ordering. This is an important measure of the resource requirements of a program. In general, it is not possible to determine the token storage requirement of an entire program in advance, as this is equivalent to the classical halting problem. However, it is possible to determine the token storage requirements for reasonably large portions of programs; this is non-trivial as there are generally many legal execution orders. (4) The resource requirements of a program are extremely sensitive to the execution order. The active portion of the invocation tree determines the amount of resources required by the program. Under sequential evaluation, at most one branch is active at any time. Under parallel evaluation the active portion may include the entire tree. It is trivial to construct examples where a maximally parallel evaluation has exponentially larger resource requirements than a sequential evaluation. The resource requirements of a program can be controlled by constraining the amount of exposed parallel activity.

The Tagged Token Dataflow Architecture

The Tagged Token Dataflow Architecture is described in Chapter 3. It is a scalable multiprocessor comprised of numerous processing elements (PEs), each a complete dataflow computer. The PE is heavily pipelined and extremely tolerant to communication latency [5]. The Tagged Token Dataflow Architecture captures the basic execution mechanism of the U-interpreter. Data is transferred between instructions as tokens, but tokens carry fixed size *tags* in place of unbounded activity names. The tag specifies the instruction which is to be executed, including the processing element that holds the instruction, and the address where the instruction resides within that processing element, An associative *waiting-matching* store is employed to retain tokens while they await their partners and to detect when instructions are enabled. When matching tokens are detected they are purged from the waiting-matching store, and the corresponding instruction is scheduled for execution. A computation is distributed over the PEs using a two-level scheme. Code-block invocations are assigned dynamically to collections of processors, called *domains*, by the run-time system. The individual activities (*i.e.*, instances of instructions) within a code-block invocation are distributed over PEs in a domain by hashing the tag [2]. Certain resources are associated with each code-block invocation, including: a code-block register, a block of program memory, and varying amounts of token storage.

In examining the Tagged Token Dataflow Architecture there are a number of important points to observe. (1) Program graphs must be embellished to engage the resource management system when resources are to be allocated and to inform this system when resources can be released. (2) If the waiting-matching store of an individual processor becomes full, the processor will deadlock. Thus, the load on the waiting matching store should be considered along with other resources when assigning work to processors, even though token storage is allocated and released implicitly. (3) The iteration number portion of the tag *will* overflow. Thus, iteration numbers must be reused like other resources. (4) If programs are allowed to unfold in an unconstrained manner, as in the U-interpreter, many programs which should execute within the resources provided by the machine will exhaust the machine resources and fail because too much parallelism is exposed. By constraining how programs unfold, it is possible to reduce their resource requirements, while still fully utilizing the machine.

The Resource Management Problems

Chapter 4 articulates the basic resource management problems in realizing the U-interpreter on the Tagged Token Dataflow Architecture. These problems can be largely overcome through sophisticated resource management. (1) Program graphs should be embellished so that a request is sent to the run-time system whenever resources must be allocated. The final token generated by a code-block invocation should be a signal to inform the system that the invocation has completed and the corresponding resources can be released. (2) The run-time system should ensure that no waiting-matching store is over-committed. This requires determining the worst-case token storage requirements of code-block invocations in advance, so the system can avoid assigning an invocation to a collection of processors with insufficient token storage. (3) Loop code-blocks should be transformed so that iteration numbers can be reused. If iterations numbers are reused properly, the size of the iteration field in the tag limits the number of concurrent invocations of a loop, not the total number of iterations. (4) The breadth of the active portion of the invocation tree should be constrained so that enough parallelism is exposed to fully utilize the machine, without requiring more resources than necessary. Following these guidelines will allow programs of significant size to execute effectively on the Tagged Token Dataflow Architecture. The latter chapters of the thesis develop the solutions suggested by this guideline, starting with a simple class of programs and progressing through more complex program structures.

Data structures present an entirely different class of resource management problems, many of which are extremely difficult. This thesis does not deal with the management of data structure storage at all. A caveat ought to be placed around each statement concerning resource requirements; data structure storage requirements are not included. Many program structures which have bounded resource requirements excluding data structure storage, may have unbounded structure storage requirements. The structure storage requirements of a program depend heavily on the model of structures employed, whether it be Dennis's general structures [15, 9] or I-structures [11][1]. The U-interpreter is defined in terms of general structures. The Tagged Token

[1]General structures require a new structure, with a single new element, be created when an element is appended. They are usually implemented as a linked structure, so that parts of the structure can be shared. I-structures are a special case of structures, requiring that each element be written at most once. They are usually implemented as an array of slots, with special hardware to allow reads to arrive before writes [17].

Dataflow Architecture supports only I-structures, as they are more efficient, but less general. The work presented in this thesis is pertinent with either form of structures. The discussion assumes the general form of structures.

Acyclic Blocks Without Conditionals

Chapter 5 examines the simplest class of programs, acyclic blocks without conditionals. The overall structure of these programs is independent of the input data and can be determined in advance. Recursion is excluded *a priori*, for without conditionals it would never terminate. Nonetheless, this class of programs is quite interesting. For a given computation, there are many (possibly exponentially many) legal execution orders, and the different orders have vastly different resource requirements. The particular order that a given machine will follow is extremely hard to predict; instructions are scheduled for execution dynamically based on the arrival of data. The arrival patterns are effected by the machine configuration, the assignment of work to processors, the structure and behavior of the network, the collisions encountered in the network, the mix of instructions, etc. Rather than try to predict the particular execution order, we will determine the worst-case resource requirements in any legal execution order.

The primary resource management problem is token storage. Unless the load placed on the waiting-matching stores of the various PEs is accounted for, an individual PE may become over-committed and deadlock, even though the overall resources in the machine are sufficient to support the computation. To account for this load, the potential token storage requirement of a code-block must be determined in advance. A powerful technique for determining the worst-case token storage requirements for such programs is developed in Chapter 5. The basic idea is to formulate the space of legal configurations of a dataflow graph as a system of integer linear constraints. The token storage requirement of a code-block, as a function of the configuration, can be maximized over the feasible region determined by the constraints. The constraint systems that arise through this approach have a particularly simple form (the dual of a min-cost flow problem) and can be solved efficiently (*i.e.*, in polynomial time). This technique is quite general, and allows for a variety of extensions.

Acyclic Blocks With Conditionals

Chapter 6 extends the constraint system technique to handle conditional expressions. This pushes the limits of the approach. Determining tight bounds on the token storage requirements of acyclic blocks with conditionals is NP-complete. However, approximate bounds can be determined efficiently and successively refined using a branch-and-bound approach. While, determining the resource requirements of an entire program employing conditional expressions and recursion is equivalent to solving the halting problem, tight bounds on the resource requirements of individual code-block invocations for such programs can be computed efficiently, for most code-blocks encountered in practice.

Cyclic Blocks

Chapter 7 focuses on the class of cyclic graphs arising from the loop constructs. These graphs present a variety of inter-related problems. (1) The iteration number portion of the tag will overflow on modest loops. (2) The token storage requirement for loops may be unbounded. (3) Loops may spawn an unbounded number of concurrent, subordinate code-block invocations. The key to solving these problems is to control the number of concurrent iterations of a loop. The main result presented in Chapter 7 characterizes the class of loops which have bounded unfolding and those which can potentially unfold into arbitrarily many concurrent iterations. A loop has bounded unfolding if and only if the graph representing the body of the loop forms a single strongly connected component. This result suggests a technique for augmenting cyclic graphs so that they have bounded unfolding. With slight extensions to the basic model, the degree of unfolding can be set at the time the loop is invoked. These augmented graphs produce the same results as the original graphs, but have more predictable resource requirements. The token storage requirements can be determined using the constraint system technique, as a function of the number of concurrent iterations. Iterations identifiers are recycled in these augmented graphs automatically; if the loop has a maximum of k concurrent iterations, then the first iteration is guaranteed to be complete when the $k+1^{st}$ iteration begins.

The analysis and transformation of cyclic graphs is addressed first for the case where conditional expressions within the loop are excluded. The results are extended to handle conditional expressions.

Dynamic Control

Chapter 8 addresses how the unfolding of general program structures can be dynamically controlled. The Tagged Token Dataflow Architecture tends to allow the invocation tree to unfold in a breadth-first manner. This approach exposes as much parallelism as possible, but causes extremely large resource requirements. A depth-first unfolding exposes less parallelism, but has smaller resource requirements. By allowing the mode of unfolding to be determined dynamically, based on the machine status, it is possible to execute large programs effectively on the machine. The program is allowed to unfold in a breadth-first manner until sufficient parallelism is generated to fully utilize the machine. At which point, a depth-first unfolding can be pursued on a number of independent branches.

Chapter Two

The U-interpreter

The U-interpreter is an abstract model of parallel computation developed by Arvind, *et. al*, [9, 3]. This chapter describes the model in detail and draws attention to those aspects most germane to resource management. The U-interpreter is independent of any particular machine. It is defined abstractly in terms of propagating data values, in the form of *tokens*, through graphs. Parallelism is implicit; the model simply relaxes the constraints on the order in which operations are performed. Tokens carry an *activity name*, in addition to a data value, which specifies the arc on which the token resides and the firing of the corresponding instruction in which it will participate. An individual firing of an instruction is termed an *activity*. The basic operations are defined by rules for generating result values and activity names based on the input values, input activity names, and the program graph. The structure of dataflow graphs is dictated by a collection of simple graph schemata[2]. The schemata play a prominent role in the latter chapters of the thesis, for they are the basis for analyzing the structure of program graphs. The schemata are defined such that all legal executions produce exactly the same results. Thus, programs are determinate, even though the order in which activities are performed is not. Synchronization of parallel operations is inherent in the mechanism for enabling operations, because operations are enabled for execution when their operands are available. The model is simple and elegant, but unrealistic for a direct implementation because it assumes unbounded queuing of tokens on the arcs, permits activity names of unbounded size, and allows an unbounded number of operations to execute simultaneously.

This chapter presents a precise, but not completely rigorous, formulation of the U-intepreter.

[2]The graph schemata presented here differ slightly from those presented in the references. Notably, the *L* operator is eliminated from the loop schema.

However, before going into detail we will develop intuition for the nature of dataflow computation. A dataflow program is a directed graph of operations. As an example, Figure 2-1 shows the graph for $(a^2 + 1) * (a^2 - 1)$. Values are transferred between operations along the arcs as tokens. An operator may execute (or *fire*) whenever tokens are available on each of its input arcs. In Figure 2-1, operation S_1 is enabled to execute. Upon firing, it consumes a token from each input arc and produces a token on each output arc. Thus, in Figure 2-1, after S_1 fires, both S_2 and S_3 will be enabled. These two operations may execute in parallel, or either may precede the other. Dataflow operators are functional in the sense that the outputs of an activity are entirely determined by the values carried on the input tokens. The pair of values received by S_4 is not affected by the order in which S_2 and S_3 fire. S_4 will not be enabled until both S_2 and S_3 have completed.

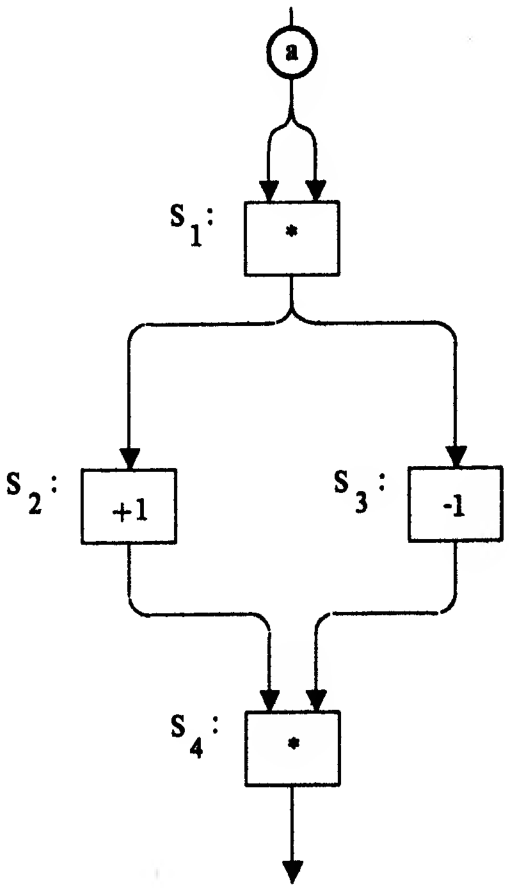

Figure 2-1: A Simple Dataflow Graph

The preceding discussion treats a dataflow program as an acyclic graph, which can be applied to a set of arguments by placing initial tokens on the input arcs. We want to extend this basic model to allow for cyclic graphs and user defined functions. With cyclic graphs it is possible for a number of tokens to reside on an arc simultaneously. The simple firing rule given above does not specify how tokens from the various arcs should be selected to participate in a given activity. Additional constraints are required to insure that the selection is completely determined. If there is any indeterminacy in the selection of tokens to participate in an activity, program results will be indeterminate as well.

Historically, these additional constraints have been introduced into the dataflow model in three forms. The static-dataflow model proposed by Dennis [15] places the constraint that no more than one token can reside on an arc. The Q-interpreter [8] allows unbounded queueing on the arcs, but requires that FIFO order be maintained; this is difficult to implement in practice. The U-interpreter allows unbounded queueing on the arcs with no imposed order; instead, each token carries an *activity name* which uniquely identifies the portion of the computation to which it belongs.

In the U-interpreter, the firing rule is extended as follows: an operator may fire whenever a token is available on each of its input arcs, such that the set of input tokens have identical activity names. Any sequence of activities which obeys this rule is a *legal execution order.* Most code-blocks will have numerous legal execution orders, since the firing rule defines a partial order on the execution of activities in a program. The exact structure of activity names and the rules for generating the activity names for result tokens are described below.

User-defined functions introduce additional complications. Intuitively, such a function is represented by a dataflow graph. Applying the function involves placing tokens on the input arcs of the corresponding graph and allowing them to propagate through the graph to produce results. The subtlety lays in how tokens are conveyed to and from the representative graph. One approach [21] suggests an *apply* operator which receives a function value and an argument value and replaces itself with the graph representing the function. In essence, function application is graph expansion. This approach encounters serious difficulties if cyclic graphs are permitted. A second approach [15] suggests that the *apply* operator forward its argument value to the graph which represents the function, with additional "uniqueness" information carried on the tokens to differentiate between distinct invocations. In essence, the program graph is fixed, but some portions have many independent waves of tokens flowing through them. The latter approach is employed in the U-interpreter, with part of the activity name (the *context*) providing uniqueness.

Activity names do not, in and of themselves, guarantee that the selection of tokens to participate in an activity is completely determined. If two tokens could be placed on an arc with the same activity name, either could be selected to participate in an activity. Legal dataflow graphs are structured so this can not happen; no two activities have the same activity name[3]. Legal dataflow graphs are defined by a collection of graph schemata and rules of composition. The schemata ensure that all potential activities in the execution of a program are assigned unique activity names. The graphs are *self-cleaning, i.e.*, no tokens remain in the graph when the program terminates, and all legal executions produce exactly the same results.

2.1. Activity names

The activity name is a 4-tple, <CONTEXT, CODE-BLOCK NAME, STATEMENT NUMBER, ITERATION NUMBER>. This is generally abbreviated <U.C.S.I>. The code-block name and statement number together identify a specific node in the program graph. A dataflow program comprises a collection of separate graphs called *code-blocks*; these correspond to individual loops or procedures in a high-level dataflow language. Each code-block is given a unique name and the operators within a code block are given unique statement numbers. The context and iteration number together identify a particular firing of the specified node. The context specifies a particular invocation of the named code-block. The iteration number specifies a particular iteration, within this invocation. Assume the graph schemata are structured so that each potential activity in a dataflow program has a unique activity name. Then the unique context for a code-block invocation can be simply the activity name of the *apply* activity which generated the invocation.

[3]There are many situations where non-deterministic behavior is required, *e.g.*, real-time systems. This can be captured in the dataflow model with specific non-deterministic operators, such a *merge*. Although non-deterministic constructs have been proposed for the U-interpreter (*cf.* the references [9, 4]), these constructs will not be considered in the thesis.

2.2. Basic Firing Rule

The basic firing rule is illustrated in Figure 2-2. Tokens carry a value, an activity name, and a port number. All tokens destined for instruction *s* in code-block *P* carry an activity name of the form ⟨U.*P*.*S*.I⟩. The different instances of this operation are differentiated by the U and I parts. All tokens participating in an activity must carry identical activity names. Thus, in Figure 2-2, the tokens carrying values of 1 and 2 enable operation S, but those carrying 1 and 5 do not. The port number specifies the input arc upon which the token resides. *An activity is enabled for execution when tokens with identical activity names are available on each port of the node specified by the activity name.* These matching tokens are consumed and result tokens are produced for the output arcs. Figure 2-2 shows the transition caused by a single firing of the + operation.

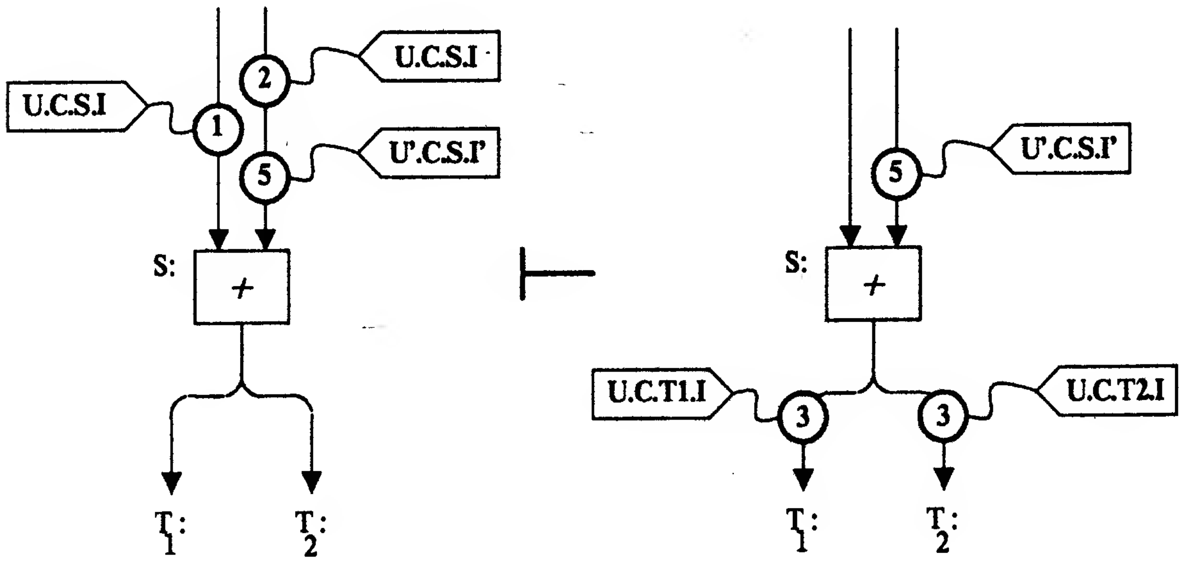

Figure 2-2: Firing an Operator

The firing rule specifies the yields-in-one-step relation for the U-interpreter. A *configuration* is simply a set of tokens. Since each token identifies the arc upon which it resides, a set of tokens is effectively an assignment of tokens to arcs in a graph. In a given configuration a collection of activities are enabled for execution. Some number of them fire, generating a new token set by removing input tokens and producing output tokens. The initial configuration of a dataflow program has a single token on the input arc of the top-level code-block.

Definition 1: A *configuration of a dataflow graph P* is a set of tokens with activity names specifying arcs in *P*.

For configurations C_1 and C_2, we say $C_1 \vdash C_2$, iff C_2 is generated from C_1 by the firing of one or more activities enabled in C_1.

A *legal execution order of a dataflow program P* is a sequence of configurations (C_1, C_2,... , C_k) such that C_1 is an initial configuration and $C_i \vdash C_{i+1}$, for each i from 1 to k-1.

A *legal configuration* is a configuration that appears in some legal execution order.

Note the U-interpreter is a non-deterministic model of computation ($\vdash$ is a relation of the form *token set* × *program graph* × *token set*), since many activities may be enabled in a given configuration. This is not surprising, as parallel computation is a mild form of non-determinism. We will we require that dataflow graphs be structured such that the results of a computation are determinate, even though the execution order is not.

2.3. Well-behaved Graphs

For arithmetic and logical operations the context, code-block name, and iteration number on the result tokens are identical to those of the input activity name. The result tokens are destined for operations within the same iteration of the same invocation of the same code-block as the input tokens. The s part and the port differ for each result token; they specify the arcs that the token is to traverse. The behavior of these operations is given by:

$\{\langle U.C.S.I, v_j\rangle_j : j = 1, ..., p\} \vdash \{\langle U.C.T_j.I, OP_s(v_1,...,v_p)\rangle_{port_j} : j = 1, ..., d\}$,

where OP_s is the operation specified by node s,

p is the arity of OP_s, and

T_1, ..., T_d are the addresses of the successor nodes.

Data structure operations (*Append* and *Select*), follow this basic rule as well. Data structures are considered values and are (conceptually) carried on tokens. *Append* takes three inputs, a structure, and index, and a value, and produces a new structure which differs from the input structure only at the specified index, where the input value is appended. The old structure is not modified. *Select* takes as input a data structure and an index and produces the value associated with the specified index.

An operator is *well-behaved* if each firing consumes a token from each input arc and produces a single token on each output arc, carrying the U, C and I common to the input tokens. Arithmetic, logical, and data structure operators are well-behaved.

Acyclic graphs of well-behaved operators are essentially like well-behaved operators; a single set of inputs produce a single set of outputs, with the same U, C and I. Consider, for example, the graph in Figure 2-1, above. We observed that this graph has three legal execution orders. S_1 must fire first. Then S_2 and S_3 may fire in parallel, S_2 may fire before S_3, or S_3 may fire before S_2. Finally, S_4 fires. With all three orders, a single wave of tokens propagates through the graph, although the respective wavefronts differ. Each activity is assigned a unique activity name, independent of the execution order, because only the S part changes, and no statement number is repeated. Eventually, a single wave of tokens is produced on the output arcs, with the U, C and I of the input wave. This property is formalized below.

> **Definition 2:** A graph is *well-behaved* if a set of tokens, one on each input arc, with identical U, C and I: (1) produces a token on each output arc carrying the same U, C and I, (2) leaves no other tokens in the graph, and (3) assigns to each activity generated by the set of inputs a unique activity name, independent of the execution order.

Acyclic interconnections of well-behaved operators are well-behaved, since the U, C and I parts of the activity name do not change and each operation fires exactly once. By induction on the depth of the graph, the results produced by an acyclic graph are determined entirely by the values carried on the input tokens, independent of the execution order.

2.4. Conditional Schema

The graph schema for conditional expressions is depicted in Figure 2-3. It employs a special operator, *switch*, which is not well-behaved. A *switch* receives two inputs, an arbitrary data value and a boolean control value, and routes the data value to one or the other output, as specified by the control value. The behavior of the switch is given by:

$$\{\langle U.C.S_{\text{SWITCH}}.I,\ v\rangle_{\text{data}},\ \langle U.C.S_{\text{SWITCH}}.I,\ b\rangle_{\text{ctrl}}\} \vdash \begin{cases} \{\langle U,C,S_{\text{TRUE}},I,\ v\rangle_{p_{\text{true}}}\}, & \text{if } b = \text{true}, \\ \{\langle U,C,S_{\text{FALSE}},I,\ v\rangle_{p_{\text{false}}}\}, & \text{if } b = \text{false}. \end{cases}$$

The conditional schema represents an expression of the form, `if P(a₁,...,aₘ) then`

F(x_1,...,x_k) else G(x_1,...,x_k), where F represents the 'true' block and G the 'false' block. The k switch nodes are controlled by a single predicate. The two sub-blocks of the conditional schema may be any well-behaved graphs. The *merge*, denoted by ⊗ is not a true operator; it denotes that two arcs converge on the same port. Since the arms of the conditional are well-behaved, a single wave of input tokens produces a single wave of tokens on either the 'true' or 'false' side, with the original U, C and I. Thus, each merge receives exactly one token, and it carries the original U, C and I. So, the conditional schema is well-behaved.

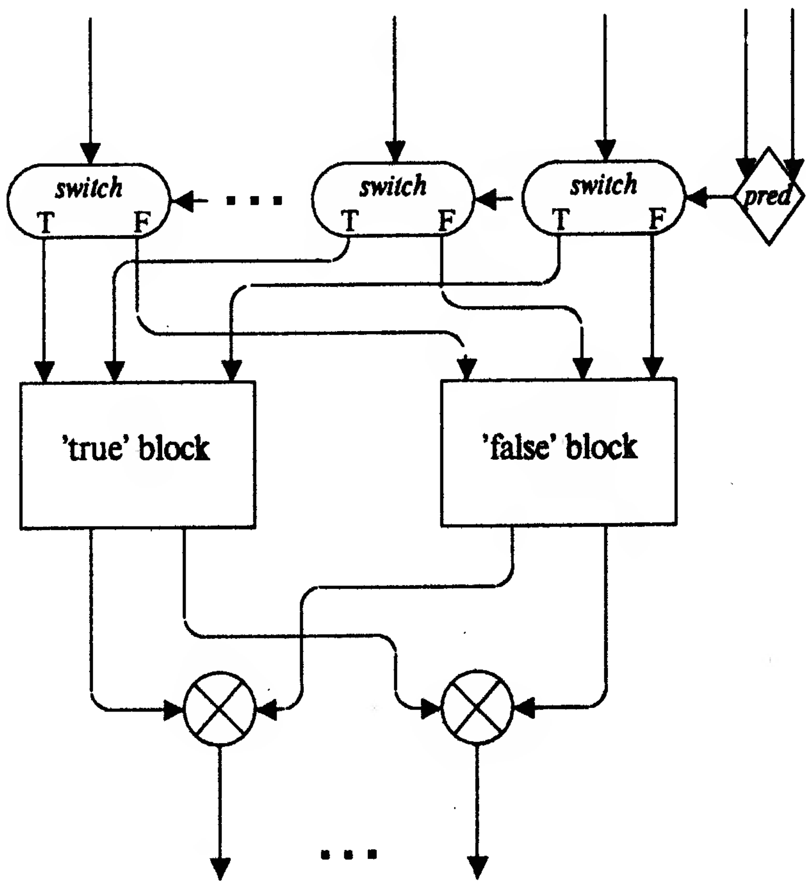

Figure 2-3: Schema for Conditional Expressions

2.5. Acyclic Code-blocks

Acyclic graphs can be encapsulated as a *code-block* to support user-defined functions. This makes the model fully general, since all partial recursive functions can be expressed in terms of function invocation and conditional expression. Function invocation changes the nature of the model dramatically. For acyclic graphs without function invocation, the computation is completely described by the program graph. Each operation fires exactly once. With function invocation, the program effectively expands whenever a function is invoked. An executing dataflow program is an extremely dynamic entity.

An acyclic code-block is simply an acyclic graph with a *begin* operator as the input node and an *End* operator as the output node (*cf.* Figure 2-4). This corresponds to a user-defined function containing no loops. The *begin* and *end* operators manipulate the context portion of the activity name in conjunction with the apply schema, as shown in Figure 2-4. The *apply* operator takes as input an argument structure and a code-block identifier. It produces a result token which is destined for the *begin* operator of the specified code-block and carries the argument structure as data. The activity name of the corresponding $apply^{-1}$ operator is stacked into the context $U' = \langle U.C.S_{APPLY^{-1}}.I\rangle$ to provide a unique context for the new invocation. The context serves also to provide a return activity name when the invocation completes. The iteration number is immaterial, and is set to zero. The *begin* operator simply passes the argument structure on to the body of the invoked code-block. Since data structures can be passed on tokens, a single argument token and a single result token is fully general.

If the body of the invoked code-block is well-behaved, the *end* operator receives a single token with context U' and iteration number 0. It unstacks the activity name for the $apply^{-1}$ operator, and thereby returns the result value to the apply schema. The $apply^{-1}$ operator simply passes the result value to its successors.

The behavior of the *apply* and *end* are given by,

$$\{\langle U.C.S_{APPLY}.I,\ v\rangle_{arg},\ \langle U.C.S_{APPLY}.I,\ P\rangle_{proc}\} \vdash \{\langle\langle U.C.S_{APPLY^{-1}}.I\rangle.P.S_{BEGIN}.0,\ v\rangle\},\ \text{and}$$

$$\{\langle\langle U.C.S_{APPLY^{-1}}.I\rangle.P.S_{END}.0,\ v\rangle\} \vdash \{\langle U.C.S_{APPLY^{-1}}.I,\ v\rangle\}.$$

The *begin* and $apply^{-1}$ are simply identity operators.

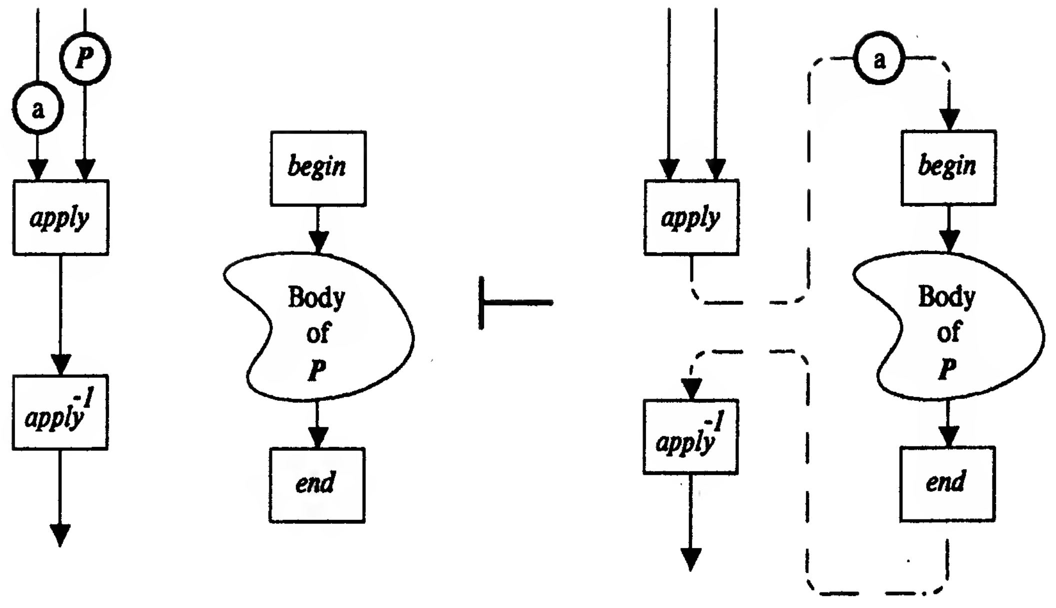

Figure 2-4: Code-block Invocation

Assuming the invoked code-block is well-behaved, the apply schema is well-behaved. So any acyclic interconnection of basic operators, conditional schemas, and apply schemas is well-behaved, assuming the invoked code-blocks are well-behaved.

The manipulation of activity names is best understood by considering the entire *invocation tree* generated by a computation. Note that if a code-block is invoked with a context U, all tokens generated by the invocation carry this context. The program is initiated when some code-block *P* is invoked with a null context <>. Each activity generated by this invocation has a unique activity name of the form <<>.*P*.S.0>. Thus, each subordinate code-block invocation, receives a unique context of the form <<>.*P*.S.0>. For example, in Figure 2-5, operation V within code-block *P* invokes code-block *Q*. Operation S' within *Q* generates another invocation, and so on. The program unfolds as a tree of code-block invocations as indicated by Figure 2-5.

By induction, each activation provides its children with a unique context. Therefore, each activity is assigned a unique activity name. Assuming the program terminates, all branches of the invocation tree are finite. The leaves generate no further invocations; thus, they are well-behaved

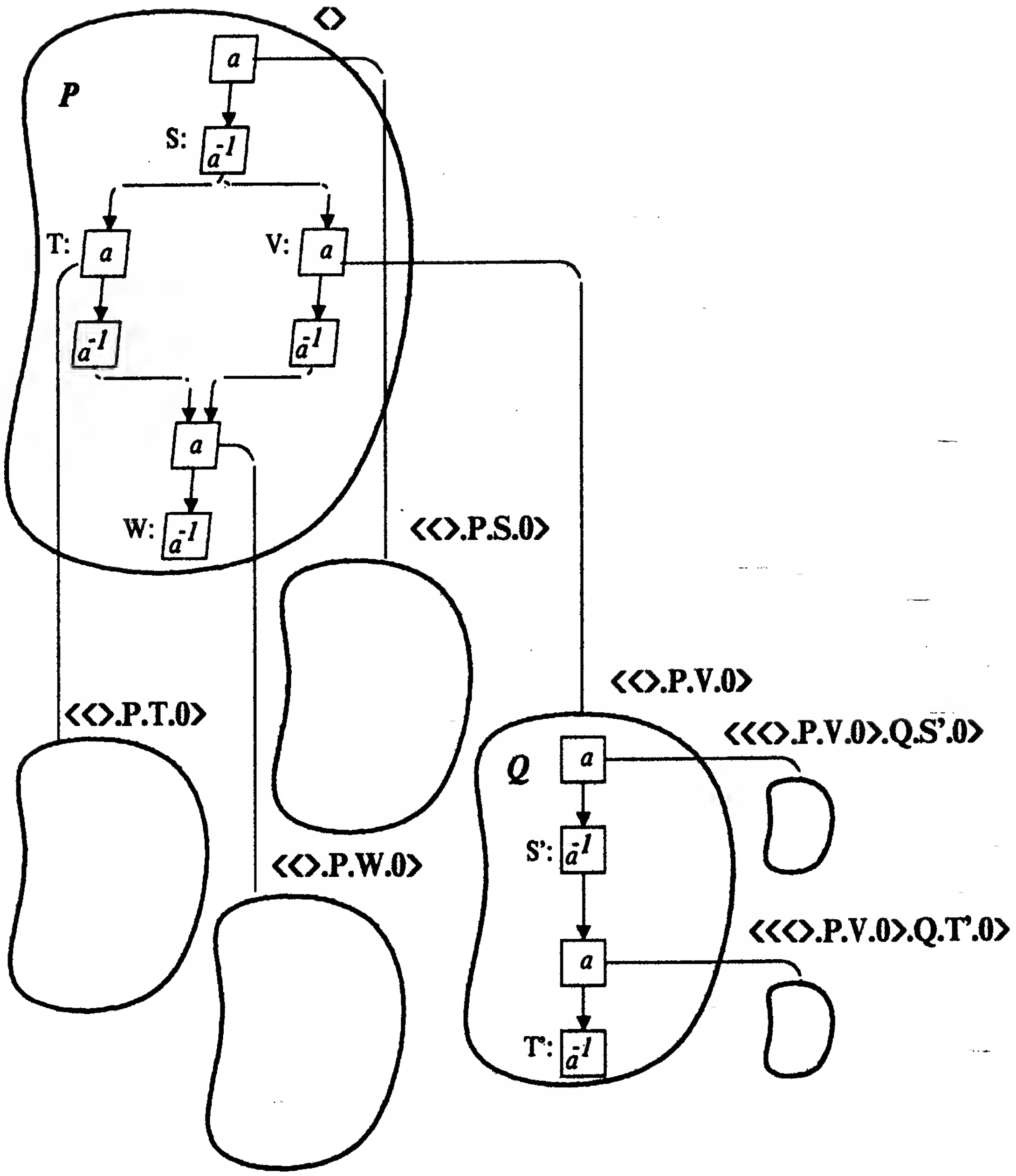

Figure 2-5: Tree of Code-block Invocations

and return tokens to their parents. By induction, each of the *apply*$^{-1}$ operators receive a single result and every code-block activation is well-behaved. Therefore, the entire program is well-behaved. Hence, all terminating dataflow programs comprised of acyclic code-blocks, of well-behaved operators, conditional schemata, and apply schemata are well behaved.

2.6. Loop Code-blocks

Iterative or tail-recursive constructs are an important special class of computations which should be carried out efficiently. These constructs are represented conveniently by the loop code-block schema depicted in Figure 2-6. This is the only form of cyclic graph allowed in dataflow programs. A loop code-block, like the acyclic code-block, has a *begin* operator, an *end* operator, and a well-behaved internal graph. However, the internal graph of a loop-code block has substantial structure. As suggested by Figure 2-6 it consists of four acyclic subgraphs, plus feedback arcs. A *loop variable* is associated with each feedback arc. The header, trailer, and loop body may be arbitrary acyclic graphs. The predicate may be any acyclic graph producing a single boolean output. The results of the header provide the first wave of tokens to the *switches* and the loop predicate. The output of the predicate is connected to the control input of each of the switches. In Figure 2-6 the one-ended arcs incident on the *switches* should be interpreted as outputs from the predicate. The initial wave of tokens have I = 0. If the result of the loop predicate is 'true', the wave of tokens is routed to the loop body. Eventually, it produces a wave of tokens on the outputs of the loop body. The *D* operator increments the iteration number in the activity name, leaving the data unchanged. Tokens circulate through the loop body until the loop predicate turns 'false'. All tokens belonging to iteration k have I = k. When the loop predicate turns 'false' it causes the final wave of tokens to be routed to the D^{-1} operators. D^{-1} sets the iteration number to 0 and passes the data on to the trailer. The behavior of the *D* and D^{-1} is given by:

$$\{\langle \mathrm{U.C.S_D.I},\ v\rangle\} \vdash \{\langle \mathrm{U.C.S.I{+}1},\ v\rangle\},\ \text{and}$$

$$\{\langle \mathrm{U.C.S_{D^{-1}}.I},\ v\rangle\} \vdash \{\langle \mathrm{U.C.S.0},\ v\rangle\}.$$

Note that loop variables need not circulate in clearly defined waves; some may circulate faster than others. The relative rate at which loop variables circulate is constrained only by the data dependencies. The iteration number specifies the iteration to which a given token belongs, so there can be no interaction between distinct iterations.

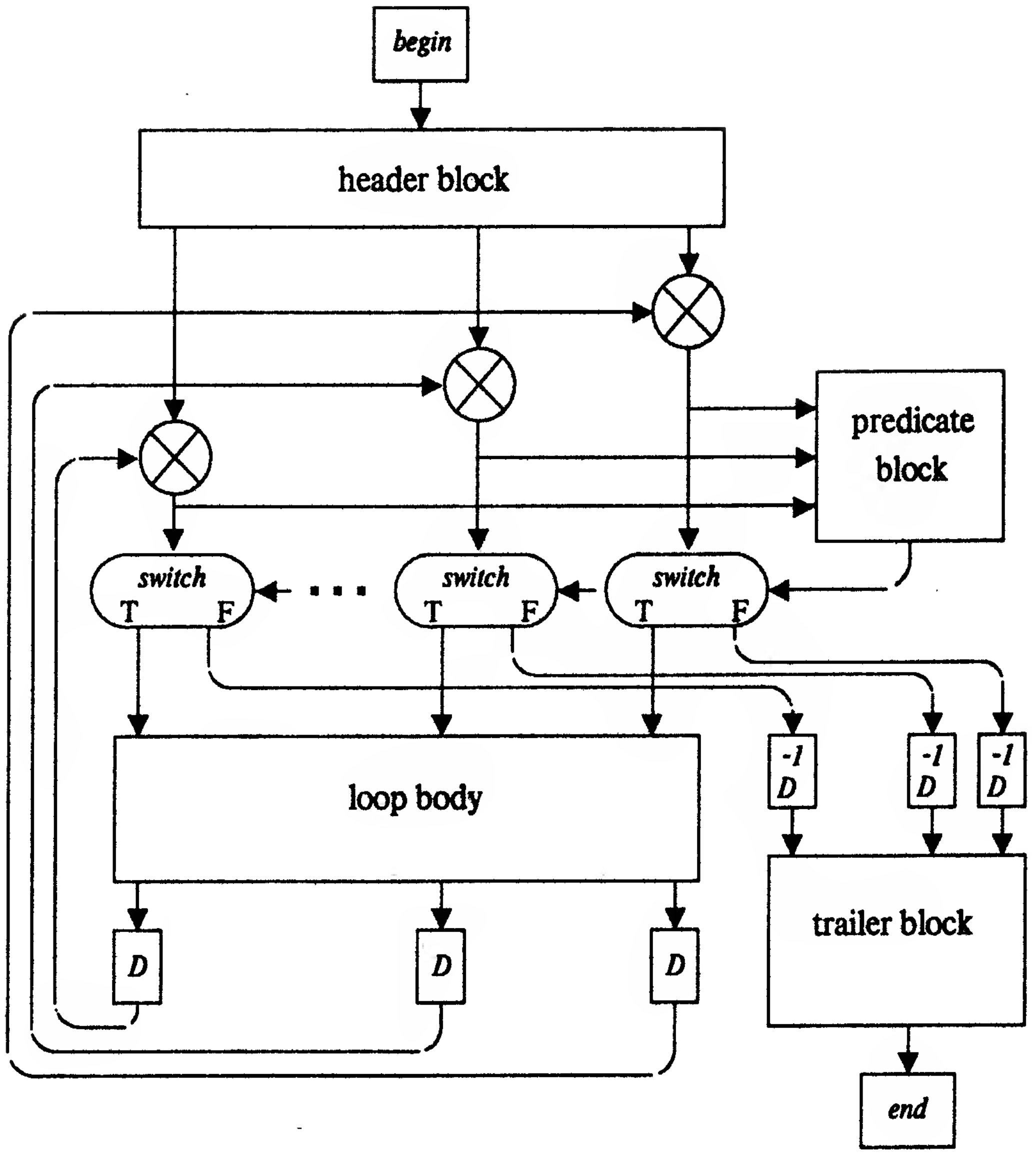

Figure 2-6: Loop Code-block Schema

As a simple example, consider the program to compute $\sum_{i=0}^{N} F(i)$, shown in Figure 2-7. The index variable I circulates and initiates N activations of F. SUM circulates, accumulating partial sums. We can assay the behavior of this program, assuming fair scheduling of operations. If F requires a long time to compute, I will circulate substantially faster than SUM. In this case, many distinct invocations of F will execute in parallel. A large collection of 'true' tokens and a single 'false' token will queue on the control input to the switch operator for SUM. As the invocations of F complete, SUM will circulate and eventually produce a result. Note that the activations of F need not complete in the order of initiation. They will, however, be summed in order, since the iteration number on the token produced by F must match with that on the token for SUM, in order to enable the + operation.

Suppose, on the other hand, F requires very little time to compute. Then, the summation offers very little parallelism. The index variable can only generate a few invocations of F before the first invocations terminate. Only a few invocations of F will be in execution at any time, and I will lead SUM by only a few iterations.

The analysis above relies heavily on the assumption of fair scheduling. The class of legal executions under the U-interpreter allow for unfair scheduling, as well. At one extreme, all N iterations of the index variable may complete, and then all N instances of F may execute in parallel, regardless of the time require to compute F. At the other extreme, the two loop variables, I and SUM may circulate together; one iteration completes before the next begins. Both of these schedules are valid under the U-interpreter, regardless of the computational requirements of F.

2.7. Dynamic Structure of Dataflow Programs

The static structure of dataflow programs is dictated by the graph schemata described in the preceding sections; a dataflow program is simply a collection of code-blocks, related by code-block invocation. This section examines the dynamic structure of dataflow programs in execution. The invocation tree is a key concept, for it provides a way to visualize the manner in which a program unfolds. By examining the invocation tree we can glean how resource allocation is embodied in the abstract model and how the resource requirements of a program are affected by the execution order.

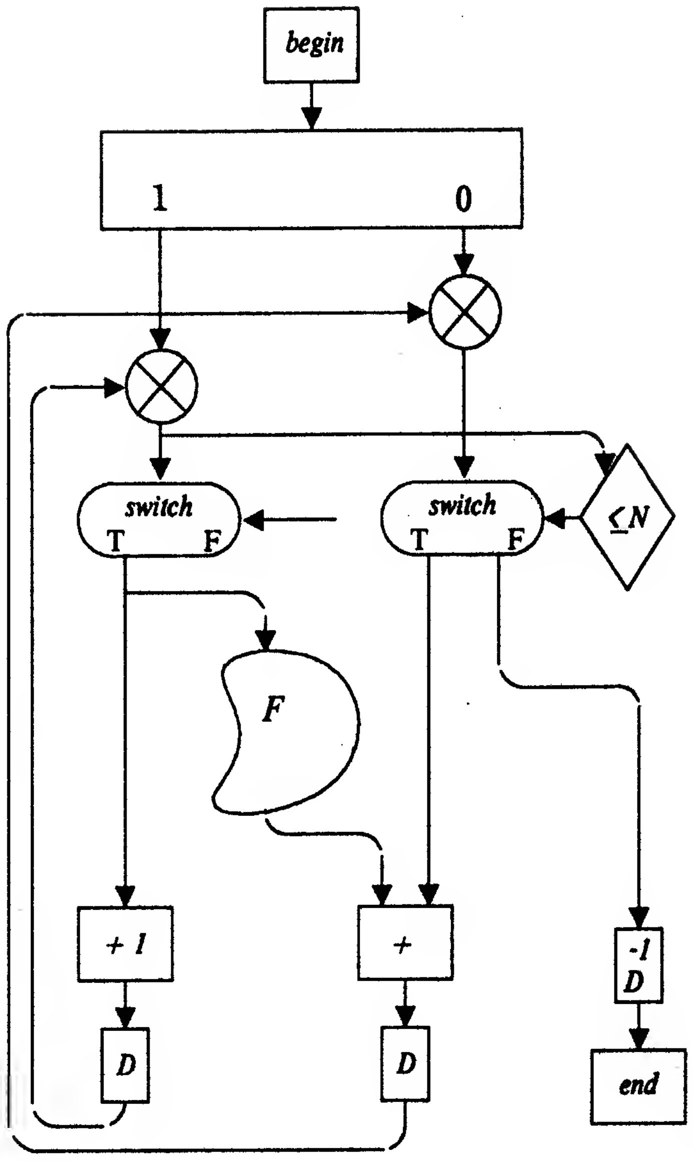

Figure 2-7: Graph for $\sum_{i=0}^{N} F(i)$

Section 2.5 described the invocation tree, assuming code-blocks to be acyclic. The observations made in that section continue to hold when loop code-blocks are included. A program unfolds as a tree of code-block invocations. Each invocation is assigned a unique context using the activity name of the corresponding *apply*$^{-1}$ activity. Each activity in the computation is assigned a unique activity name, denoting its position in the invocation tree. The invocation tree, and the activity names assigned within it, are independent of the execution order, *i.e.*, all execution orders of a given program on a given input produce the same invocation tree. The primary difference introduced by loop code-blocks is that a code-block invocation may represent a large, dynamically determined quantity of computation, since an invocation may involve many iterations. Each iteration may generate subordinate invocations, so the branching of the tree may be large and dynamically determined. The various subordinate invocations generated by a loop code-block are distinguished by the iteration number of the parent buried in their contexts.

As an example, consider the computation depicted in Figure 2-8. Code-block *P* invokes two code-blocks *Q* and *R*. *R* is a loop code block, and each iteration invokes *T*. Each invocation of *T* is assigned a unique context of the form $\langle\langle\langle\rangle.P.S_P.0\rangle.R.S_R.I_R\rangle$.

The invocation tree unfolds as a program progresses. At any time, only a portion of it (a subtree) is active; some portions have completed; others have not yet be initiated. From a resource management viewpoint, the active subtree is extremely important. A branch comes into existence when a code-block is invoked and disappears when it terminates. The active subtree describes the current state of the computation. At any time, the resource requirement of a program is precisely the resources required to support the active subtree.

Code-block invocation is the primary resource allocation operation in the U-interpreter. Abstractly, a code-block invocation requires a context, a certain amount of token storage, and a certain amount of processing support. It represents a block of computation which will execute independently of its parent until it completes. Throughout its execution there will be tokens residing in the graph. A number of activities will be generated and performed. A context is allocated by simply extending the activity name; since activity names are unbounded, this can be done in an elegant, local fashion. Token storage and processing support is implicit; these resources are also unbounded.

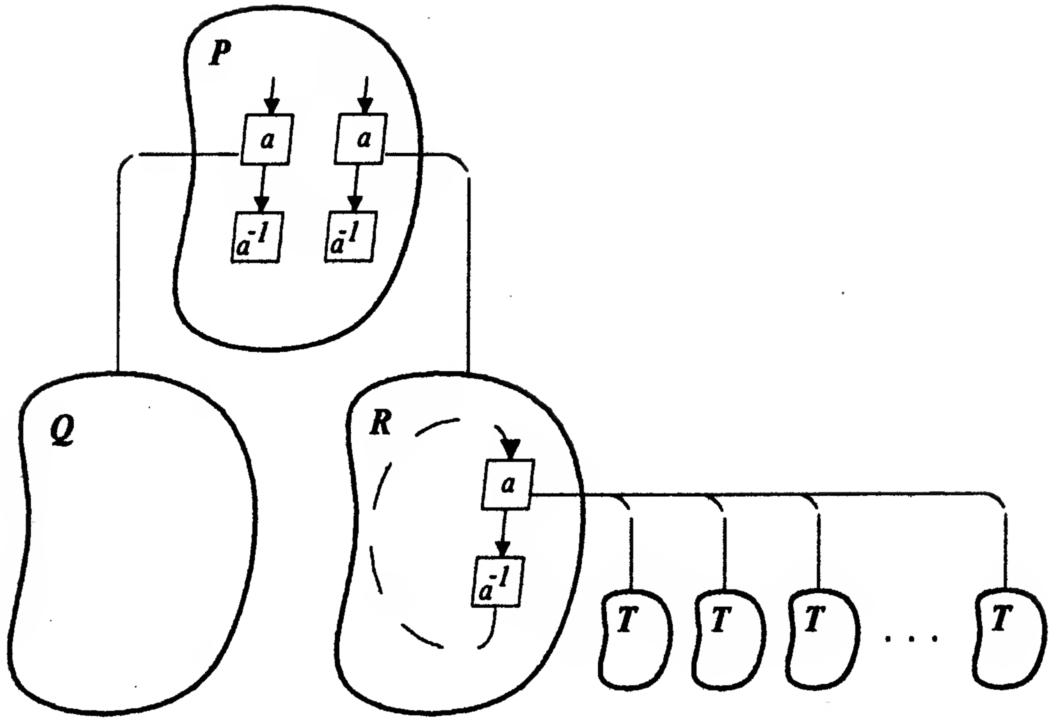

Figure 2-8: Invocation Tree with Loop Code-blocks

Even though the overall invocation tree is independent of the execution order, the size of the active subtree (and hence the resource requirements) over time is extremely sensitive to the execution order. Consider the example, in Figure 2-8. The subtree rooted at *Q* may be active before, after, or in parallel with the subtree rooted at *R*. If these subtrees represent substantial blocks of computation, the resource requirements of the program will depend heavily on whether these two subtrees are active concurrently, or not. The resource requirements of a program can be regulated by restricting the breadth of the active portion of the invocation tree, but this limits the amount of exposed parallelism as well. The breadth of the active subtree in a particular execution order corresponds closely with the amount of exposed parallelism.

2.8. Summary of the U-interpreter Model

A dataflow program, under the U-interpreter, is a collection of loop and acyclic code-blocks, related through code-block invocation. The internal structure of these code-blocks is restricted by a collection of graph schemata to ensure that programs are determinate. In general, there are many legal execution orders for a program; they offer differing amounts of parallelism, but all produce the same results. The individual operations of a dataflow program are functional in the sense that their results depend only on their inputs; they have no internal state and no side-effects. A program in execution unfolds as a tree of code-block invocations. Each invocation is assigned an unique context by extending the activity name of its parent. Each activity is assigned a unique activity name, which specifies its position in the invocation tree.

Code-block invocation is the primary resource allocation operation. A new context is generated, and all resources required to support the invocation are implicitly provided. All tokens generated by the invocation will carry the same context. The token storage requirement of an invocation is simply the maximum number of tokens that co-exist carrying the associated context. This is extremely dynamic and depends on the particular execution order that is followed. Processors do not enter into the definition of the formal model directly; the model is defined in terms of propagating tokens through graphs. Each activity implicitly requires processor support, so we might define the processing requirement of a code-block invocation as the number of activities generated by the invocation.

Activity names are potentially unbounded. The context portion grows linearly with the depth of the invocation tree. The iteration number grows logrithmically with the number of iterations performed by a loop code-block invocation. Generating new contexts by stacking the activity name performs two essential functions: it provides a unique context for the subordinate invocation, and it provides a return activity name. The growth of activity names can be reduced if these two properties are separated. Since the invocation tree is indeed a tree, as opposed to a general graph, never do two independent activities need to generate the same *new* context. The context for a new invocation be simply a unique identifier, with no further semantics. This requires that the return activity name be passed explicitly to the subordinate invocation. With this approach, the context part grows logrithmically with the size of the invocation tree. If contexts are reused the size of the context is logarithmic in the size of the active subtree. Note, this requires detecting when invocations terminate and releasing the associated contexts. Extending the activity name a allows contexts to be generated in a completely decentralized fashion; retaining this property with the unique identifier approach requires care.

The resource requirements of a program depend on how the invocation tree unfolds. If we assume fair scheduling, the tree unfolds in a breadth-first manner. The active subtree grows very broad, very quickly. Assuming an infinite number of processors and no communication delay, this greedy schedule is optimal. It generates a tremendous amount of parallelism. However, the resource requirements of a computation are determined by the size of the active subtree becomes during the computation. A less eager approach which limits the amount of exposed parallelism by unfolding branches in a depth-first manner requires significantly less resources. In a practical realization of the model, a limit must be placed on the number of concurrent activities. Thus, to allow large programs to execute on the machine, it may be necessary to limit the breadth of the active subtree.

Chapter Three

The Tagged-Token Dataflow Architecture

The Tagged Token Dataflow Architecture is a multiprocessor based on the U-interpreter, under development by the Functional Languages and Architectures Group at the Massachusetts Institute of Technology. It consists of a collection of processing elements (PEs), each a complete dataflow computer, connected via a packet switched communication network. The machine is intended to exploit the diffuse, unstructured parallelism common in general purpose computation. Also, it is intended to be truly scalable; performance should improve with the simple addition of processing elements. Parallelism is exploited by allowing independent activities to execute on different PEs. The dataflow scheduling mechanism provides automatic, instruction level synchronization of parallel computations. This mechanism also allows a dataflow processor to operate efficiently in spite of long, unpredictable communication latencies. A processing element does not pause, as a conventional processor might, when an instruction requires data from a distant processor or memory module; it continues to execute other enabled instructions. The instruction requiring external data will be scheduled whenever the data arrives. Independent threads of computation are interlaced in the instruction pipeline; this makes it possible invest a certain amount of parallelism in masking communication latency [10]. The essential component of this architecture is an associative waiting-matching store, which detects when instructions are enabled.

This chapter describes the Tagged Token Dataflow Architecture in detail. The basic machine organization and operation is presented, demonstrating how the U-interpreter firing rule is realized. The a variety of higher-level issues are examined, including: the distribution of work over the machine, the structure of tags, and the resources associated with a code-block invocation.

3.1. Basic Organization and Operation

The basic organization of the Tagged Token Dataflow Architecture is depicted by the block diagram in Figure 3-1. The boxes represent pipeline stages; they operate asynchronously and are connected via FIFO buffers. A processing element contains three subsystems: instruction processing pipeline, data structure store, and PE controller. The stages of the instruction processing pipeline reflect the basic steps in the execution of a dataflow instruction: detect when input tokens are available, fetch the instruction, perform the operation, generate result tags, and finally dispense result tokens. A data structure operation (*e.g.*, *Append* or *Select*) causes a request to be sent to the structure storage controller responsible for the appropriate data element. The PE controller supports input/output, diagnostics, and resource management operations.

Data is passed between instructions as information packets, *i.e.*, tokens, carrying a data value and a *tag*. Tags function like activity names, with some important differences. An activity name specifies the position in the invocation tree of a particular activity; by definition, all tokens destined for a given activity carry the same activity name. No two activities have the same activity name. A tag must specify where the match is to be performed, in addition to identifying an activity. To this end, a tag carries four items of information: the address of the PE where the activity will take place, the address within that PE of the instruction to execute, a context identifier, and an iteration identifier. The size of the tag is fixed; they are not allowed to grow as the invocation tree unfolds. All tokens destined for a given activity must carry the same tag. However, over the course of a computation, a given tag may be associated with many different activities.

Upon arriving at a PE, a token enters the *waiting-matching* section. The tag it carries is compared against the tags of each of the tokens resident in the waiting-matching store. If a match is detected, the pair of matching tokens is purged from the waiting-matching section and forwarded to the instruction processing pipeline. Instructions are limited to one or two operands. For a monadic instruction, no match is required, so the input token bypasses the waiting-matching section. For a dyadic instruction, a single match must be performed to enable the activity.

The machine language of the Tagged Token Dataflow Architecture is essentially an adjacency list representation of dataflow graphs. Each instruction specifies an operation code and a list of addresses, representing successor instructions to which results should be sent. The *instruction-fetch*

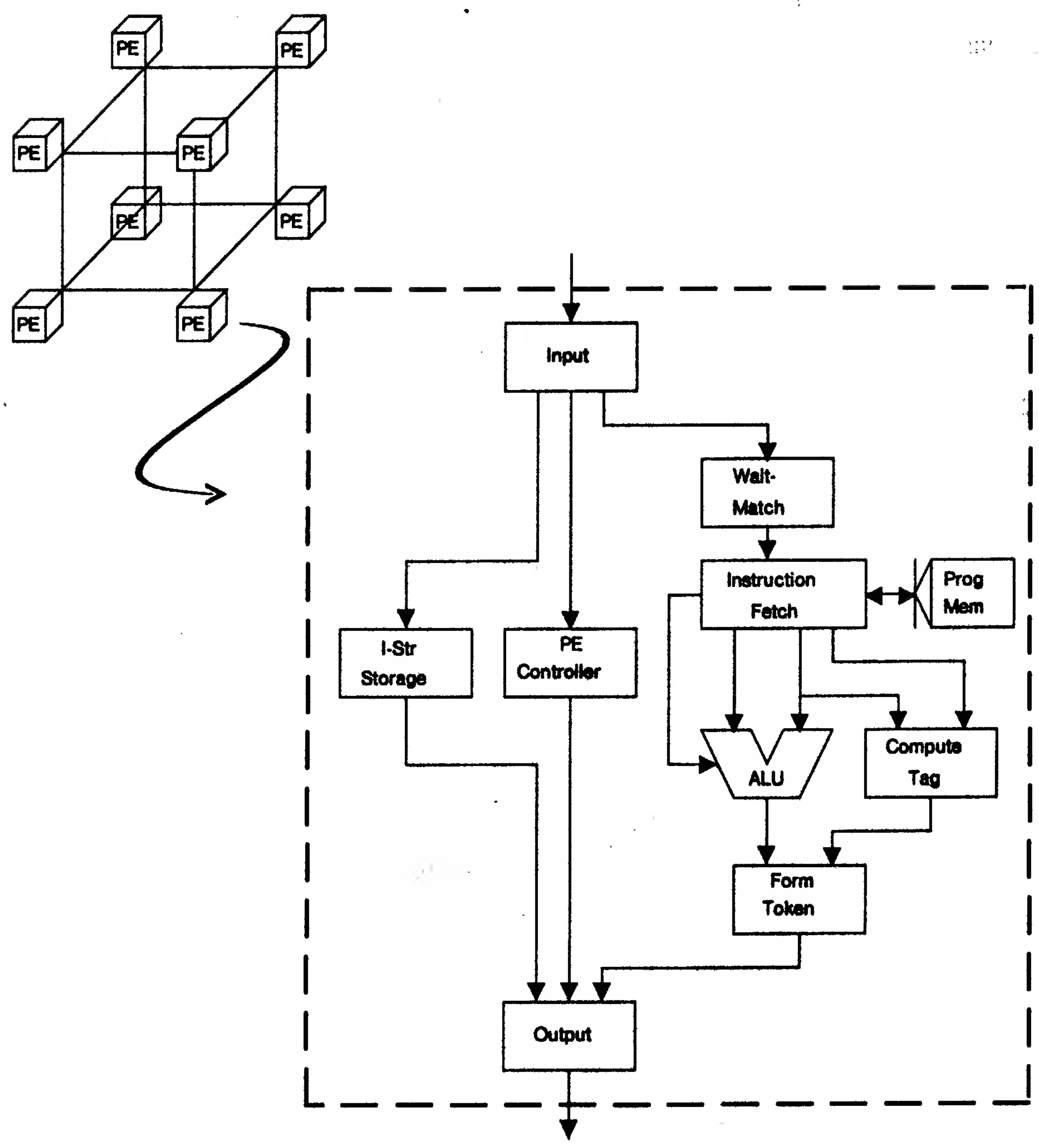

Figure 3-1: Organization of the Tagged Token Dataflow Architecture

section reads the op-code from the address specified in the tag, fetches whatever constants are required, aligns the operands, and sends the op-code and data to the *ALU* for processing. It also sends the instruction address and data to the *compute-tag* section. The compute-tag section operates in parallel with the ALU, determining the tag for each of the destination activities. Results and tags are merged in the *form-token* section, and tokens are delivered to the *output section* to be dispensed to the network. They will be routed to the PEs address specified in their tags.

Data structure operations are processed in two steps, first by the instruction pipeline, then by a structure controller. Consider, for instance, a *select* instruction. It is enabled when it receives two inputs, a descriptor for a structure and an index. The ALU computes the address of the selected element, while the compute tag section constructs a tag for the activity which is to receive the result. The result tag can not be used immediately, because the data resides in a possibly distant structure store. The element address and the result tag are sent to the appropriate structure storage controller. The structure storage controller reads the specified element and forms a token contain this data value and the result tag accompanying the request. The result token is dispensed to the network and directed to the successor of the *select* activity.

3.2. Operating Assumptions

The fundamental operating assuption in this architecture is that a token arriving at the waiting-matching section must be permitted to enter; otherwise the machine will deadlock. This potential hazard should be fully understood. Token storage is managed directly by the hardware. Storage is allocated whenever a token enters the waiting-matching section and fails to find a partner. It is deallocated when a matched pair of tokens is purged and passed on to the instruction processing pipeline. This resource presents a serious deadlock potential; it is in finite supply, non-preemptable, and acquired incrementally [14].

Suppose the waiting-matching store is completely full of tokens awaiting their partners. What happens when an additional token arrives at the waiting-matching section? It cannot be ignored, because it may match with a token waiting in the store; the only way for storage to be made available is for matches to occur! What happens if this new token does not find a match? It can not be added to the store. It can not be destroyed. Sending it back out into the network to return at a later time only postpones the problem. The collection of code-block invocations executing on the PE are deadlocked; each has acquired a portion of the token store and requires more in order to complete its execution.

We might assume a certain amount of inexpensive (non-matching) overflow store, so that in this circumstance the incoming token could be put aside. At best, this postpones the problem. If the overflow store is full, the deadlock occurs. The PE may come to a halt even if the overflow store is not full. We assume that the overflow store has no matching capability; otherwise it would be part of the matching store. If a pair of matching tokens both reside in the overflow store, the match will go undetected. Tokens can only be matched if they reside in the matching store simultaneously. We might try to fix the problem by searching linearly through the overflow store when a new token arrives or by shuffling tokens randomly between the matching store and overflow store, but these approaches fail to address the real problem. The token storage resources are simply over-committed.

A variety of dataflow machine similar to the Tagged Token Dataflow Architecture employ a two-level matching store. A small, fast, associative store and a larger overflow store with limited matching capability. When the associative store becomes full, tokens are placed in the second level store. If a token arrives and fails to find a partner in the associative store, the second level store is searched, either by scanning through the entire store or by tracing down a linked structure. In either case, performance degrades dramatically when the first-level store is full. The deadlock arguments above pertain if both stores become full. Performance considerations dictate that in fact the first-level store should not be over-committed.

There is a second, more subtle deadlock hazard arises from the handling of matched tokens and tokens which do not require partners[4]. The description of the machine operation in Section 3.1 implies that whenever a match is found the pair of tokens are purged from the waiting-matching store and forwarded to the instruction processing pipeline. Tokens which do not require a partner bypass the waiting-matching section entirely. Such a policy will cause the PE to deadlock, unless token buffers essentially the size of the matching store itself are provided. The FIFO buffers between stages in the Tagged Token Dataflow Architecture are intended to be small (two to five tokens); their purpose is to absorb variations in packet flow caused by irregular operation times and interactions with the network. They are not intended to store large numbers of tokens.

[4]To the author's knowledge, this potential hazard has not been discussed at all in the literature.

The number of matched tokens and tokens which require no partner is exactly the number of simultaneously enabled activities. This depends on the nature of the program and the manner in which it unfolds, not the relative speeds of the hardware components. The number of simultaneously enabled activities in a PE is a measure of the amount of parallelism exposed in the invocations executing on the PE. This may grow extremely large, essentially as large as the total number of tokens in co-existence. As soon as the number of simultaneously enabled activities exceeds the amount of token buffering in the PE, the PE will deadlock.

To understand this problem at a concrete level, the PE can be viewed as essentially two components: a storage and scheduling unit (S-unit) and an execution unit (E-unit). These are connected by FIFO buffers, as shown in Figure 3-2. They may be internally pipelined, or whatever. The E-unit provides little or no storage for tokens. The buffer on the left contains *activity packets* for enabled activities (*i.e.*, matched token pairs and tokens which need no partner). The buffer on the right contains result tokens which have not yet been considered for matching. In order for the activity packet buffer to remain small, the E-unit must perform activities at the same rate as the S-unit produces activity packets. In order for the result buffer to remain small, the S-unit must receive tokens as fast as the E-unit produces results. During phases of the computation in which the amount of exposed parallelism is expanding, the rate at which activities are enabled exceeds the rate at which activities are performed. Data for enabled activities builds up in the buffers, assuming no matched pairs are retained in the S-unit. If the E-unit can process activities as fast as the S-unit can enable activities, the result buffer will fill up. Once the result buffer is full, the E-unit must wait for the S-unit to process results. This causes the activity packet buffer to begin filling up. The situation is reversed if we assume the S-unit can enable activities as fast as the E-unit can produce results. Regardless of the relative speeds of the two components, once the number of concurrently enabled activities exceeds the amount of buffering the processor deadlocks; each unit requires the other to remove a packet from an input buffer in order to remove another packet itself.

To avoid deadlock, any cycle of data paths in the Tagged Token Dataflow Architecture must include an effectively infinite buffer. *i.e.*, a buffer which will never become entirely full. Thus, in the remainder of the thesis we assume the machine to be modified to include (i) a token buffer within the input section to hold surplus tokens, except those requiring partners, and (ii) a facility within the waiting-matching section to hold surplus matched token pairs.

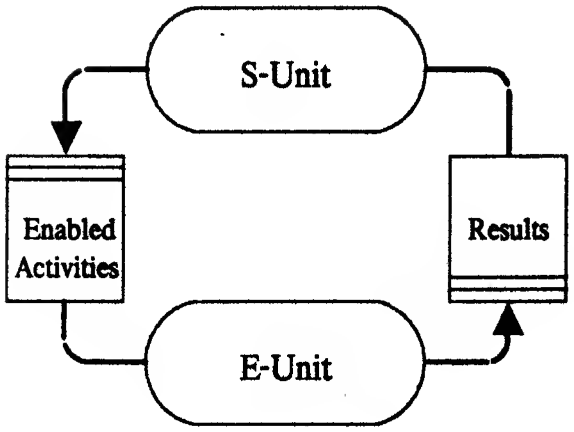

Figure 3-2: Basic PE Components

3.3. Code-block Invocation

The machine operation, described above, involves some important assumptions. First, when a token arrives at the waiting-matching section of a PE, the instruction it is destined for must be resident in the local program memory. Second, tags must be generated for result tokens, using only the input tag and the local processor state, such that all tokens destined for a particular activity receive the same tag. The assignment of activities to processors is completely outside the abstract model, as is setting up processors to support a given assignment. Note that in the U-interpreter, activity names can be generated using only the input tokens and a description of the graph. The situation is somewhat more complicated in the Tagged Token Dataflow Architecture because tags are of fixed sized, coupled with the allocation of resources, and coupled with the distribution of activities. To understand how these issues are addressed in the Tagged Token Dataflow Architecture, it is necessary to understand the run-time structures associated with code-block invocations and the structure of tags. This section outlines the process of code-block invocation and the associated run-time structures. The next section describes the structure of tags and how they are generated.

In the U-interpreter, invoking a code-block merely involves assigning it a new context. All computational resources are assumed to be unbounded, so the resources required to perform the invocation are allocated implicitly. Processors are not specifically involved. In the Tagged Token Dataflow Architecture, a code-block invocation is explicitly assigned to a collection of processors. These processors must provide all the computational resources required to support the invocation. Collectively, they must have a copy of the graph; they must provide storage for tokens generated by the invocation; and they must perform the activities it comprises. The manner in which PEs co-operate to perform a code-block invocation is an essential aspect of the Tagged Token Dataflow Architecture.

Processor domains

If processing elements are allowed to cooperate on a code-block invocation in arbitrary ways, it becomes difficult to coordinate resources. For example, if program memory becomes fragmented, it may be difficult to find a block of available space of a given size in each of a collection of PEs. To reduce the complexity of resource allocation, a hierarchy is enforced on the resources in the machine. The PEs are partitioned into a collection of *domains*, each a set of consecutively addressed PEs. This partition is fairly static; it is fixed for entire execution of a program. When a code-block is invoked, the invocation is assigned to a domain. Only the PEs in the assigned domain will co-operate to perform the invocation. Resources are allocated uniformly across a domain, so there is no difficulty in co-ordinating resources. Resource allocations in separate domains are completely independent, and there is never any need to co-ordinate them.

When a code-block invocation is assigned to a domain, a code-block register (CBR) is allocated to the invocation. The register number serves as a context identifier, as suggested at the end of the preceding chapter. It uniquely identifies the invocation, until the invocation terminates; whereupon the identifier (and register) may be reused[5]. A code-block invocation is assigned the same CBR throughout the domain, as suggested by Figure 3-3. The code-block register is loaded with the base address of the of the code-block associated with the activation.

Program memory is allocated uniformly throughout a domain as well. Suppose an invocation of an acyclic code-block l bytes in length is assigned to a domain of m PEs. Then $\lceil l/m \rceil$ bytes are

[5]This description offers a slight simplification of the Tagged Token Dataflow Architecture, as it has been presented in the past [6]. Colors and color-block areas have been eliminated, giving the machine a cleaner structure.

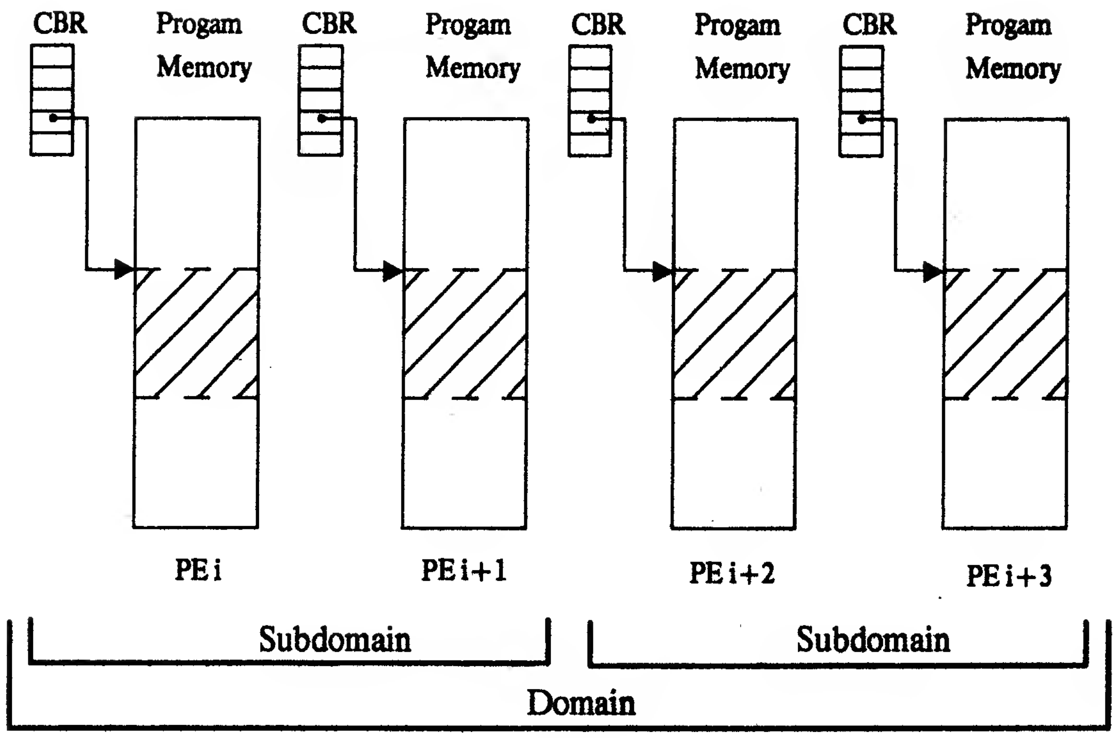

Figure 3-3: Allocation of Program Memory

allocated in each PE, starting at the same base address in each. The code-block occupies a rectangular window across the domain, as suggested by Figure 3-3. Co-ordinating resources in this manner is trivial; a single memory map pertains to every PE in the domain. The code-block register records the base address, the domain size (m), and the code per PE ($\lceil l/m \rceil$).

Local distribution of activities

The distribution of the activities comprising an invocation of an acyclic code-block is dictated by the distribution of the instructions which form the program graph; each activity executes in the PE in which the corresponding instruction resides. Each instruction in the graph fires at most once per invocation. Since the code-block is represented by an adjacency list of the graph, instructions may be listed in any order. Each instruction specifies the relative address of its successors, within the code-block. Those instructions listed in the first $\lceil l/m \rceil$ bytes are executed by the first PE in the domain, those in the next $\lceil l/m \rceil$ bytes are executed by the second, and so on. Thus, the distribution of activities comprising a code-block invocation across a domain is statically determined by the order in which instructions are listed in the representation of the graph. Care is required to assure that no instruction is split when the graph is partitioned.

Loop code-blocks allow two degrees of freedom in distributing activities across a domain: activities within an iteration may be distributed across a collection of PEs, and different iterations may be assigned to different collections of PEs. A domain is divided dynamically into disjoint *subdomains*, each a set of consecutively addressed PEs. A copy of the code-block is allocated across each subdomain, in the manner described in Figure 3-3. Note that the code per PE is $\lceil l/m' \rceil$, where m' is the number of PEs per subdomain. Iterations are dynamically assigned subdomains based on iteration number; the first *k* iterations are assigned to the first subdomain, the next *k* to the next, and so on, wrapping around when the last subdomain is reached. The partitioning into subdomains and the number of iterations per subdomain may differ for different code-blocks in a domain.

For a code-block to be invoked in a given domain, a copy of the code must be loaded there. However, once it is loaded, many invocations may share the copy of the code, *i.e.*, many CBRs may reference the same copy of the graph. A CBR contains essentially the following information: base address, domain size, code per PE, subdomain size, and iterations per subdomain. The subdomain size, code per PE, and number of subdomains per domain are restricted to being powers of two so that tags can be generated by simple shifts and masks[6].

Constant Areas

One particular inefficiency with loop code-blocks is that arguments which serve as constants throughout the execution of the loop must be circulated through the body of the loop. Numerous instructions are executed just to pass these constants along. For example, in the inner loop of a matrix multiply, six of the eight loop variables are input parameters which serve as constants. Each constant passes through two instructions per iteration. The Tagged Token Dataflow Architecture offers a way to avoid circulation of constants. A loop code-block invocation has a *constant area* associated with it. Constant areas are in program memory. Constant arguments are stored in the

[6]It is actually these shifts and masks that are kept in the CBR.

constant area by the loop header. The body of the loop is not permitted to execute until the constants have been stored in the constant area of each PE in the domain. A special counter is used to detect when the constants are in place. The constant area is an additional resource associated with a loop code-block invocation.

Invocation process

The domain/subdomain structure circumscribes how resources can be allocated to a code-block invocation. The actual allocation and initialization is performed by a resource management system which maintains the status of system resources, much like a conventional operating system. The U-interpreter apply schema is replaced by a Use Schema, which engages the resource management system (the *manager*), as shown in Figure 3-4. Upon receiving an invocation request, the manager chooses a domain to perform the invocation. It allocates a code-block register and causes the code to be loaded in the domain, if it is not already present. When loading the code, it chooses the subdomain size, and the number of iterations per subdomain[7]. For a loop code-block a constant area is allocated in program memory as well.

3.4. Tags

The structure of the tag is essentially dictated by the code-block invocation mechanism described above. The tag is composed of four, fixed size fields: <PE#, CBR, Iteration Identifier, Instruction Offset>. The PE# specifies the PE which is to execute the instruction. The CBR specifies the code-block register which records the disposition of the code-block. The instruction offset gives the address, relative to the base-address of the code-block, of the instruction to execute.

The generation of result tags given the instruction, the input tag, and the CBR is fairly straight forward. An instruction specifies the relative address in the code-block of each of its destination instructions. Suppose the destination instruction address is a, and the iteration number is i. The result tag is computed as follows.

[7]The author has implemented such a resource management system as part of a detailed simulation of the Tagged Token Dataflow Architecture. This provides a basis for investigating various resource allocation policies. A variety of load-leveling policies have been implemented to distribute work over domains.

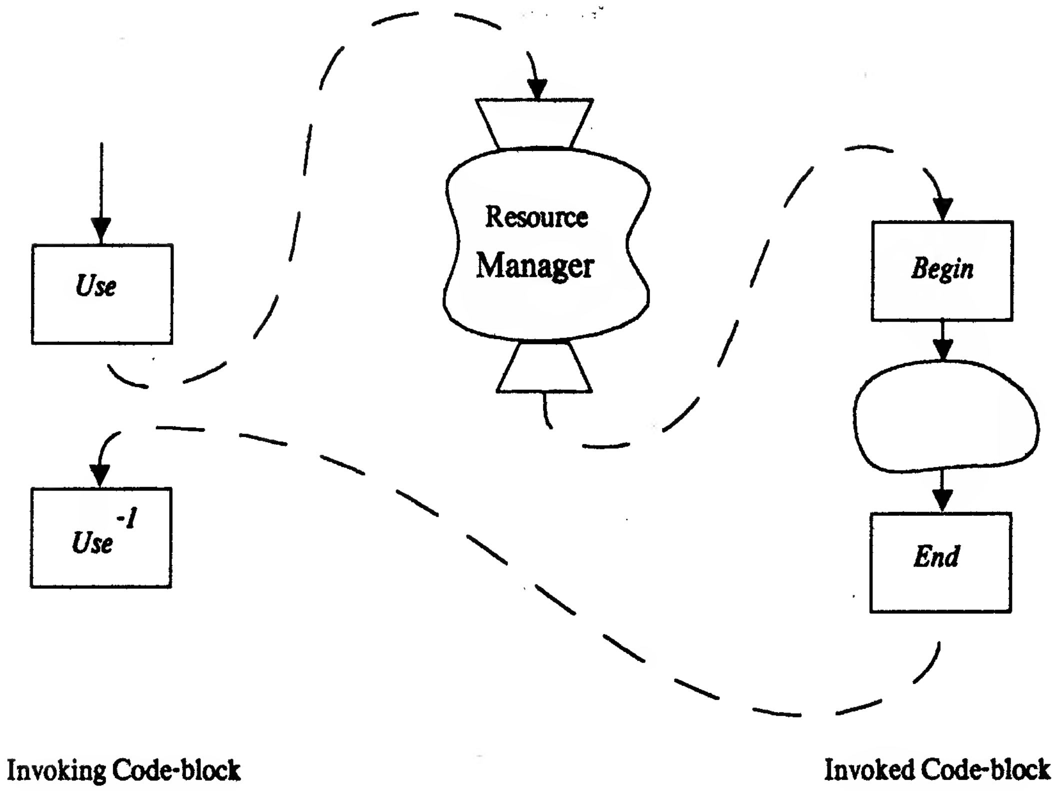

Figure 3-4: Use Schema

PE # $= PE_{base} + (\lfloor i/k \rfloor \text{ MOD } N_{sd}) * S_{sd} + \lfloor a / C_{pe} \rfloor$, and

Instruction Offset $= a \text{ MOD } C_{pe}$,

where

k is the number of iterations per subdomain,

PE_{base} is address of the base PE in the domain,

N_{sd} is the number of subdomains in the domain,

S_{sd} is the number of PEs per subdomain,

C_{pe} is the amount of program memory allocated for code in each PE.

The five constants appearing above are recorded in the code-block registers. Thus, tag generation requires the CBR and iteration identifier from the input tag, the contents of the corresponding CBR, and the successor instruction address list. Currently the tag is 44 bits in length, partitioned as follows:

PE#	8 bits,
CBR	12 bits,
iteration	8 bits, and
instruction offset	16 bits.

3.5. Summary

The Tagged Token Dataflow Architecture captures the essential dataflow instruction scheduling mechanism of the U-interpreter through the use of an associative waiting-matching store. The machine leans toward a fair scheduling of instructions, except for perturbations introduced by the vagaries of external communication. Data is passed between instructions as labeled tokens, with the label identifying the activity for which the token is destined. The nature of the labels carried on tokens is an important point of difference between the model and the machine. Activity names in the U-interpreter are potentially unbounded and identify a particular activity within the invocation tree. Tags in the Tagged Token Dataflow Architecture are bounded in size and identify the machine resources delegated to performing a particular activity. This difference is indicative of the basic difference between the model and the machine. The model is resource independent, with important assumptions of unboundedness. The machine must operate within hard and fast resource constraints.

Chapter Four

Resource Management Problems

The preceding chapters presented an abstract model, the U-interpreter, and a concrete machine, the Tagged Token Dataflow Architecture. The major difference between the model and the machine is the viewpoint adopted toward computational resources: unbounded and implicit on the one hand, bounded and explicit on the other. This chapter outlines four major resource management problems which must be addressed for large programs to execute efficiently on the Tagged Token Dataflow Architecture. The remaining chapters of the thesis provide solutions to these problems.

The essential properties of the U-interpreter model are (i) dynamic scheduling of operations based on the availability of data and (ii) automatic unfolding of programs. The first aspect is addressed in the Tagged Token Dataflow Architecture through the use of an associative waiting-matching store. The second aspect is tricky because the abstract model assumes unbounded resources, and relies rather heavily on this assumption, whereas the machine must operate within strict resource constraints. Chapter 3 outlined a strategy for dealing with this problem: introduce a resource management system which maintains the status of all machine resources. By suitably restricting the ways in which code-block invocations are distributed over PEs, it is possible to keep the complexity of managing these resources within reason.

The introduction of a resource management system addresses only part of the problem, however. Let us review the machine resources associated with a code-block invocation in the Tagged Token Dataflow Architecture. Certain resources are allocated *explicitly*: a CBR, program memory for code, and program memory for constants. The resource manager checks that sufficient quantities of these resources are available to support an invocation and allocates the required amount. Other resources are allocated *implicitly*: token storage, and tags. The resource manager need not allocate them explicitly since they are managed directly by the hardware. When an invocation is initiated in a domain, the waiting-matching stores of the PEs in the domain must accommodate the tokens generated by the invocation; the exact amount of token storage required by an invocation is not known in advance and depends on the particular execution order the invocation follows. Allocating a CBR implicitly reserves a collection of tags to the invocation; the iteration identifier may take on a range of 256 values; the instruction offset may take on a range of 16K values. These implicitly allocated resources raise potential hazards. What happens if a waiting-matching store becomes full and can not accommodate tokens for an invocation as required? What happens if a loop performs more than 256 iterations? Explicitly allocated resources raise another kind of problem. What happens if no domain offers sufficient resources to support an invocation. We may be tempted to conclude that the program is simply too large for the machine. After all, this is what we would conclude for a conventional machine if a program causes the stack to overflow. It may not be valid for a dataflow machine, however, because the resource requirements of a program depend on how the invocation tree unfolds. The real problem may be that too much parallelism is exposed. These problems are examined in detail below.

4.1. Termination Detection

The resource capacity of a machine ultimately limits the size of program that can execute on the machine. In the Tagged Token Dataflow Architecture, resources are allocated whenever a code-block is invoked. For reasonably large programs to execute on the machine, completion of code-block invocations must be detected, and the associated resources released. If the resources associated with an invocation are not released, the resource capacity of the machine dictates a limit on the overall size of the program. If resources are released and reused, the resource capacity of the machine limits the number of invocations that can be active concurrently. How do we determine when an invocation is complete? We might be tempted to claim that an invocation is complete when the *end* instruction executes; unfortunately, this is not always correct. Activity may continue within an invocation after the *end* fires. We must determine that all the activities comprising an invocation have fired.

The problem alluded to above is exemplified in Figure 4-1. If the predicate P is 'true', the output of F is used to produce the final result. If P is 'false', the output of F is discarded. Thus, generation of the final result does not guarantee that all computation within the invocation is complete. We can not rely on time-outs, or the like. The computation of F, for example, may continue indefinitely[8]. Termination must be detected explicitly, such that when the last activity of an invocation has fired a signal is generated for the resource manager indicating completion[9].

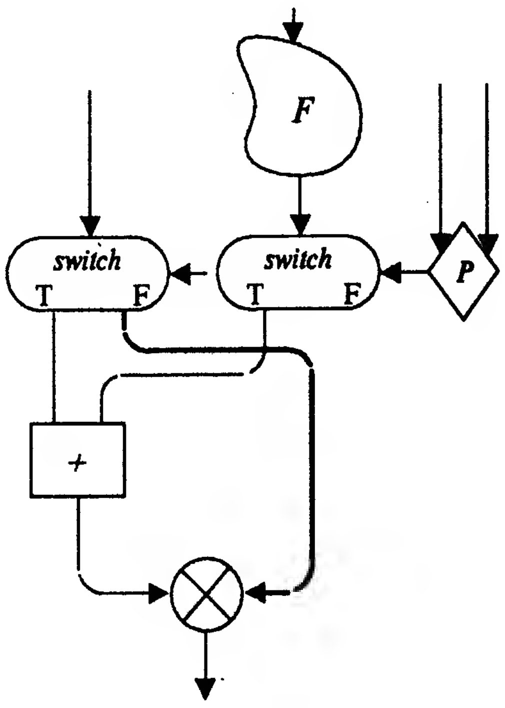

Figure 4-1: Typical Termination Problem

4.2. Token Storage Overflow

In Chapter 3 we observed that the Tagged Token Dataflow Architecture will deadlock if the facility for storing tokens overflows. In order for the machine to operate correctly it must be guaranteed that no token store in any PE ever overflows. How is this to be enforced? Observe that

[8]In practice, dataflow implementations often include operations which have side-effects, such as *write*; these may have no output arcs, since they generate no result. Thus, they also introduce problems concerning termination detection

[9]The techniques presented here for detecting termination are a refinement of those developed by Vinod Kathail, Keshav Pingali, and Arvind in implementing the Id compiler. The conditions developed here for detecting termination are needed for the other aspects of the theory to hold together.

at any point in a code-block invocation, the token storage requirement of the invocation is simply the number of tokens in existence, belonging to the invocation. At the time a code-block is invoked, if enough token storage is reserved to accommodate the maximum number of tokens that can possibly co-exist for the invocation, the token store can never overflow. This places the onus of avoiding token-storage overflow on the resource management system. A code-block invocation can not be assigned to a domain if the invocation may cause the token store to overflow. This observation has important ramifications for resource management. First, it is necessary to bound the worst-case token storage requirement of each code-block in a program in advance, *i.e.*, prior to executing the program. Second, when a code-block is invoked, the load on the waiting matching store must be taken into account in determining whether a domain has sufficient resources to support the invocation. Enough token storage should be reserved to accommodate the maximum number of tokens that could be in co-existence for the invocation, With two forms of token buffering, as suggested in Section 3.2, the load on each store must be accounted for.

Viewing token storage in this way has important ramifications in the design of the processing element as well. The size of the waiting-matching store is a fundamental design parameter for machines like the Tagged Token Dataflow Architecture; it will ultimately determine the organization of this critical component of the machine. How large this should be has been a longstanding open question in the dataflow community. There have been a variety of attempts to answer this question empirically by executing benchmark programs on real or simulated machines. In light of the preceding discussion, the answer is quite simple; it is primarily a question of balancing resources. If the PE is intended to support I simultaneous code-block invocations, the size of the waiting matching store should be I times the typical worst-case token storage requirement of a code-block invocation. For example, each PE of the Tagged Token Dataflow Architecture has 4K CBRs. So if the typical token storage requirement of a code-block invocation, loops included, is 100 tokens, storage for 400K tokens should be provided. The typical token storage requirements of code-blocks abstracted from a broad class of programs is a far more useful metric than the overall token storage requirements of a collection of benchmarks. The former metric captures a particular aspect of typical program structure, analogous to the average size of the code for a procedure. The latter reflects the happenstance of interactions within particular programs and is greatly dependent on the input data.

4.3. Tag Management

Allocating a CBR effectively reserves a 2-dimensional space of tags, delimited by the 16 bits of instruction offset and 8 bits of iterations identifier. The size of the instruction offset limits the amount of code that can be loaded into a PE for a single code-block. This limitation causes no serious trouble, because 16K is fairly large compared to the average size of a code-block, and a code-block can always be split into smaller code-blocks. The size of the iteration identifier field is a serious limitation. Many loops perform more than 256 iterations! To guarantee that this field can never overflow, it would have to be made quite large, say 40 bits. Such a large tag would introduce significant overhead. Thus, it is important to deal earnestly with overflow of the iteration identifier field. There are two possible ways to deal with this problem: extend the iteration field by allocating multiple CBR, or reuse iteration identifiers within a CBR. The remainder of this section examines the pros and cons of each approach.

Consider first the use of multiple CBRs. In many cases, the number of iterations that a loop will perform can be determined just prior to initiating the loop. FOR loops are a prime example. This information can be conveyed to the resource manager with the invocation request, allowing it to allocate a sufficient number of CBRs. The role of the *D* operator becomes somewhat complex; when the iteration number overflows, it is reset to 0 and the next CBR is used. Unfortunately, this solves the problem for only a specific class of loops. In general, the number of iterations a loop will perform can not be determined in advance. Thus, there must also be a mechanism for allocating additional CBRs dynamically.

Dynamic allocation of CBRs introduces a variety of complications. First, the role of the *D* operator becomes quite complex. If the iteration field overflows and no pre-allocated CBRs remain, a request for additional CBRs must be generated. This can be accomplished by providing two sets of outputs for the *D* operator (rather like the *Switch*), one for normal operation and one for the overflow case. One of the *D* operators in a loop is designated to generate a manager request for CBR allocation upon overflow. The overflow mode of the other *D* operators must receive the result of the allocation request before continuing.

Second, there is a potential for deadlock, if CBRs are allocated incrementally. If many loop invocations are executing in a domain, the entire supply of CBRs can become exhausted before any of the loops complete. This deadlock can be avoided if termination of iterations is detected while the loop is executing. A given iteration can depend only on earlier iterations; it can not depend on later ones. If termination of iterations is detected incrementally, the early iterations numbers can be released and reused. There must also be a mechanism for queueing a potentially large number of CBR allocation requests within the resource manager.

At this point we should consider the second alternative, reusing iteration identifiers with a single CBR, because this essentially involves detecting termination of iterations. If termination of iterations is detected, a loop never requires more than a single CBR; thus, extending the iteration field is not necessary. In the extreme case, a loop can execute given only a single iteration identifier; all activities for one iteration must complete before any of the next begin. More generally, a loop can be given a supply of iteration numbers, which are treated like tickets. When an iteration completes, its iteration number is released. An iteration number must be available for a new iteration to begin.

If iteration identifiers are reused, the size of the iteration field limits the number of iterations that can execute concurrently, rather than the total number of iterations that can be performed by the loop. If the extent to which a loop can unfold can be controlled, *i.e.*, if the maximum number of concurrent iterations can be bounded, the loop needs only a bounded number of iteration identifiers.

Reusing iteration identifiers with a single CBR appears to be the simpler approach, since no special overflow handling is required. A new iteration can begin if an iteration identifier is available. The remaining question is whether it is reasonable to limit the number of concurrent iterations to, say, 256 per subdomain. The crux of this issue is how the amount of parallelism generated by a loop compares with the amount of parallelism that a processing element can exploit. Assuming reasonably fair scheduling, for a loop to unfold into a large number of concurrent iterations, each iteration must involve a substantial block of independent computation. The amount of parallelism a processor can exploit is essentially the length of the execution pipeline[10]. Beyond the point where the pipeline is saturated, additional parallel activity serves only to increase the size of the queue of enabled activities, with no improvement in performance.

[10]It is actually somewhat greater than this because data structure operations are involve work not represented in the pipeline.

Consider, for example, the summation loop discussed in Section 2.6. Suppose computing F and accumulating the sum takes k times as long as circulating I through one iteration. Then, at most k iterations can be in execution simultaneously. By the time the $k+1^{st}$ instance of F is initiated, the first instance is complete. Thus, for this loop to unfold into 256 iterations, F must involve the equivalent of approximately 1K instructions, since circulating the index variable requires four sequential instructions.

There are two important points to be gleaned from this example. First, if a loop possesses enough parallelism to allow it to unfold into a large number of concurrent iterations, each iteration involves a substantial computation. Such a loop can keep a large number of processors fully utilized. A loop which possesses enough parallelism to allow it to unfold into 256 concurrent iterations involves enough computation to keep at least 16 PEs fully utilized. If iteration numbers are allocated per subdomain, 256 concurrent iterations per subdomain is more than ample for reasonably sized subdomains. Secondly, it is trivial to construct loops which unfold into an arbitrarily large number of concurrent iterations.

The fundamental question is not whether the iteration field should be 6, 8, or 12 bits, but rather, how can loops which could potentially unfold into more than the available number of iteration numbers be controlled to they operate within these limits. Ideally, loops should be structured so that iteration identifiers are recycled automatically; by the time the k + ith iteration is enabled, the ith iteration should have completed. This allows a loop to execute an arbitrary number of iterations using k iteration identifiers. The *D* operator simply assigns the next iteration identifier, modulo k.

As a final note, there is an issue of architectural aesthetics. The decision to have the iteration field be small enough that iteration numbers must be reused, rather than having the field be so large that it cannot be exhausted, is a strong architectural directive. It is better to exploit this directive than to circumvent it.

4.4. Program Deadlock

Each code-block invocation requires certain computational resources: CBR, token storage, storage for code, and storage for constants[11]. Each of these resources are in finite supply, and so

[11]Storage for data structures is also required, but we are excluding data structures form consideration in this thesis.

place a limit on the number of code-block invocations that can be in execution concurrently on the Tagged Token Dataflow Architecture. Assumming that termination of invocations can be detected and that token storage and iteration overflow can be solved, we should consider what happens if any one of these resources becomes exhausted. What happens if the manager receives an invocation request and no domain has sufficient resources to support the invocation? The manager might queue the request and wait for resources to be made available. Unfortunately, queueing the request may prove useless, because all active branches of the invocation tree may require additional invocations before they can begin to release resources. The various active branches of the invocation tree are competing for system resources. Each has acquired certain resources, and requires still more in order to complete its task. The program is deadlocked due to lack of resources.

As the active subtree of the invocation tree grows (*i.e.*, as code-blocks are invoked) resources are allocated. They are only released when the active subtree retracts (*i.e.*, when invocations terminate). System deadlock occurs when a program unfolds to the point where some resource is exhausted, and yet no branch of the active subtree extends deep enough to terminate and begin releasing resources. Additional resources are required to allow any branch to extend to the point where it will begin releasing resources.

In a conventional machine, one would say that such a program is simply too large for the machine. It requires more resources than the machine offers. In a dataflow machine, this claim is not valid; it is possible for a program to deadlock even though no single branch of the invocation tree requires more than a fraction of the total resources of the machine. The resource requirement increases with the amount of exposed parallelism, regardless of the amount of parallelism the machine can actually exploit. If a great deal of parallelism is exposed, the active subtree is broad and bushy. This decreases the size of program that can execute on the machine.

The solution seems clear, avoid pursuing so much parallel activity. Unfortunately, this is not so simple. Under the U-interpreter, a program unfolds in accordance with the parallelism present in the program. The Tagged Token Dataflow Architecture captures this aspect of the U-interpreter precisely. The machine provides essentially fair scheduling of activities, because enabled activities are processed in FIFO order. If the amount of exposed parallelism in the program is greater than the amount of parallelism the machine can exploit, the machine timeshares among the active threads of computation on an instruction by instruction basis. The invocation tree tends to unfold in a breadth-first manner, exposing maximal parallelism. A large program will unfold in this way until the active subtree grows so large that some computational resource is exhausted and the machine halts. Limiting the breath of the active portion of the invocation tree, allows larger programs to execute on the Tagged Token Dataflow Architecture, without exhausting the computational resources. The primary resource management problem is to control the unfolding of programs so that enough parallelism is exposed to fully utilize the machine, while still allowing very large programs to execute within the resource constraints the machine imposes.

A couple examples should help elucidate the problem. Consider a program which employs binary recursion, and suppose the recursion extends l levels. A sequential, or depth-first, execution requires at most l concurrent invocations at any instant. A breadth-first execution generates 2^{l+1}-1 concurrent invocations. Thus, if the machine has resources to support only 32K (*i.e.*, 2^{15}) concurrent invocations and the program extends 15 or more levels, a breadth-first execution will deadlock. A slightly less eager strategy will expose ample parallelism, and yet allow the program to execute to completion. The limit need not be imposed by CBRs; other resources may be constraining. A maximally parallel evaluation requires exponentially more resources than a sequential evaluation.

As a more concrete example, consider the matrix multiply program shown in Figure 4-2. It is written in the dataflow language Id [11]. The graph for this program has three code-blocks, one for each of the loops. Collectively these occupy a total of 2.5K bytes of program memory. A single copy of the code can be shared by many invocations. The constant areas are 64, 80, and 112 bytes in length for the outer, middle, and inner loops, respectively. In a fully parallel evaluation, one outer loop, n middle loops, and n^2 inner loops may be in execution concurrently. Thus, $64 + 80n + 112n^2$ bytes of program memory are required for constant areas alone. The token storage requirements is approximately $10n^2$ with fair scheduling, *i.e.*, about ten tokens per instance of the inner loop[12]. Multiplying two 16 by 16 matrices in a fully parallel evaluation requires nearly 32K

[12]With a less fair scheduling, the token storage requirement is order(n^3). Each instance of the inner loop generates n different values of k, which are only consumed as the summation progresses. Our empirical studies using a detailed simulation of the machine generate close to the $10n^2$ figure.

bytes of program memory and 2.5K elements of token storage, at approximately 10 bytes per token! A single PE is saturated by about four instances of the inner loop, depending on the relative speed of the structure memory and the instruction pipeline. By restricting the unfolding so that each PE performs say four concurrent instances of the inner loop, the resource requirements per processor are less than 1k bytes of program memory for constants and 50 elements of token storage. The program would still fully utilize the machine.

```
PROCEDURE matrixmul (A, B, n)
                                 ! Matrices represented as array of rows !
 (INITIAL C <- <>                ! Create empty structure                !
  FOR i FROM 1 TO n DO           ! Fill in the rows                      !
      C[i] <- (INITIAL Row_C <- <>         ! Start with empty row        !
               FOR j FROM 1 TO n DO        ! fill in the elements        !
                 Row_C[J] <- (INITIAL sum <- 0
                              FOR k FROM 1 TO n DO     ! inner product  !
                                NEW sum <- sum + A[i,k]*B[k,j]
                              RETURN sum)
               RETURN Row_C)
  RETURN C)
```

Figure 4-2: Dataflow Program for Matrix Multiply

System deadlock is difficult to avoid, in general, because the size of the subtree that will be generated by a particular invocation can not be predicted. However, we should not despair. The goal is not to determine in advance whether a given program will execute to completion on the machine; the goal is to allow as large a class of programs as possible to execute efficiently on the machine. By controlling the way programs unfold, it is possible to control their resource requirements to a certain extent. The class of programs that execute to completion on the machine enlarges, as unfolding is restricted. However, restricting unfolding limits parallelism. It is unreasonable to strive to execute any program for which no single branch of the execution tree exceeds the resource capacity of the machine; this would require that all the PEs act as resource servers, while a single PE does the computation. The goal is *effective parallel* computation, so we should not restrict parallelism below some multiple of the number of PEs in the machine. A reasonable goal would be: a program should execute effectively and to completion on *n* PEs if no single branch of the invocation tree exceeds the resource capacity of a single PE. The resource management problem is how to control program unfolding, in accordance with the availability of resources, to achieve this goal.

4.5. Summary

The U-interpreter is a powerful, elegant framework for describing parallel computation. It does not rely on the implicit timing properties of any particular architecture; all interactions are via explicit transfer of tokens. It is not cluttered or constrained by the limitations or idiosyncrasies of any particular machine. It removes all unnecessary controls, allowing programs to generate as much parallel activity as possible. Realizing this model on a concrete machine requires reinstating certain controls. We do not want a program to generate an immense amount of parallel activity, if that parallelism can not be exploited. We must recognize the resource limitations of the particular machine and introduces constraints that reflect these limitations.

This chapter has identified four major limitations of the Tagged Token Dataflow Architecture.

1. Termination of code-block invocations must be detected. The hardware provides no mechanism for determining when all activities for a code-block invocation have fired, so program graphs must be embellished so that a particular node in each code-block is guaranteed to be the last activity for an invocation of the code-block. A signal can then be generated to inform the resource manager that the resources associated with the invocation can be released.

2. Token storage is limited. Even though this resource is managed directly in hardware, the resource management system must take explicit care to avoid over-committing the waiting-matching store and the token buffer in each individual PE. For this to be successful, it is necessary to determine the worst-case token storage requirements of code-blocks in advance.

3. The supply of iteration identifiers per invocation is limited. We do not want to allow loop code-blocks to generate an arbitrary number of concurrent iterations. We want to limit the number of concurrent iterations and to recycle the iteration identifiers.

4. The number of concurrent invocations the machine can support is limited. We want to restrict the breadth of the active portion of the invocation tree so that just enough parallelism is exposed to saturate the PEs. This allow large programs to execute effectively on the Tagged Token Dataflow Architecture.

These resource management problems are addressed in the remaining chapters of the thesis. Chapter 5 considers a restricted class of programs, comprising acyclic code-blocks without conditionals. For this class of programs token storage overflow is the most serious hazard. A powerful technique is developed for determining the worst-case token storage requirements for this class of programs. Chapter 6 enlarges the class of programs by introducing conditionals. Again the primary issue is token storage overflow. The introduction of conditionals changes the complexity of determining token storage requirements dramatically, and we must settle for slightly loose bounds. Chapter 7 enlarges the class of programs further by considering loop code-blocks. The three resource management problems boil down to one essential problem: transform loops so that they have bounded unfolding. For these transformed loops, it is possible to predict the token storage requirements, recycle iteration identifiers, and control the breadth of the active portion of the invocation three. Chapter 8 addresses controlling the unfolding of programs in the large.

Before proceeding with the resource management problems outline above, we should note that a variety of other resource management problems arise in the Tagged Token Dataflow Architecture which are not addressed in the thesis. One of these is management of data structures. A policy must be instituted for determined how structures are to be allocated across the machine. There is potential for exploiting program structure to achieve high locality, but this will require sophisticated compilation techniques and a mechanism for conveying the results of the static analysis to the run-time system. There is also a question of avoiding structure store contention. In addition, a record of the status of the structure store must be maintained. The exact nature of this task depends to a large extent on the model of structures that is assumed: dynamic heap allocation *ala* Dennis [15] or arrays of slots [11]. The Tagged Token Dataflow Architecture supports the array of slots model. Fairly conventional dynamic memory management techniques suffice to record the availability of structure storage. There is as of yet no facility for dealing with fragmentation of the structure store. The allocation policy is still an open problem.

A second important problem is to develop an effective policy for determining where on the machine code-blocks should be activated. Again, there is potential for exploiting special program structures. The author has implemented a variety of dynamic load leveling mechanisms which appear to perform well in practice. The distribution policy is important from a performance point of view, but plays a secondary role compared to the resource management problems enumerated above.

Chapter Five

Analysis of Acyclic Blocks

To open our attack on these resource management problems, we consider programs comprised of acyclic code-blocks, without conditionals. Later chapters examine broader classes of programs, by including the other graph schemata. Acyclic blocks without conditionals form a particularly restricted class of programs. The entire invocation tree can be determined in advance, and all inputs generate the same tree. Nonetheless, the dynamic behavior of these programs, *i.e.*, the manner in which the invocation tree unfolds over time, is extremely difficult to predict. It depends on the low-level interactions which influence the execution order, such as network contention, distribution of work, instruction mix, etc. The primary resource management problem is token storage overflow. As the tree unfolds, invocations are assigned to domains. If token storage requirements are not accounted for, a PE may become over-committed and cause the program to deadlock, even though sufficient resources are available elsewhere in the system. Termination detection is straight-forward. Tag management is not a problem, because iteration identifiers are not used. Program deadlock is a potential problem, but can be dealt with fairly well in this restricted setting. Since the invocation tree can be known in advance, the total resource requirements of the program can be bounded in advance. If this bound is less than the resource capacity of the system, the program will execute to completion. This approach is rather conservative, since only a portion of the invocation tree can be active at any time. We should like to derive a tight bound on the largest subtree (in terms of resource requirements) that may be active in a legal execution order.

The token storage overflow problem and the program deadlock problem boil down to a common question, "Of the space of legal program configurations, what is the worst configuration by some metric?". Without placing any restrictions on the execution order, beyond those implied by the firing rule, we want to determine the worst configuration that may be achieved. The metric may be the number of tokens on the arcs or the number of concurrent code-block invocations.

This chapter develops an algebraic formulation of the concept of a legal configuration in terms of integer linear constraints on the number of times that adjacent nodes fire. The feasible region of these constraints corresponds with the space of legal configurations. Thus, the question asked above can be stated as an integer linear program, which for acyclic graphs without conditionals can be solved efficiently[13].

5.1. Termination Detection

Determining that all activities for a code-block invocation have fired is straight-forward for acyclic blocks without conditionals. We need only ensure that every node is on a path from the *begin* node to the *end* node. This is a reasonable requirement since dataflow instructions are enabled by the arrival of input data and have no side-effects. If a node is not on a path from the *begin*, it can never fire. If a node is not on a path to the *end*, it can not affect the result of the computation[14].

> **Definition 3:** An acyclic code-block without conditionals is *well-connected* if every node is on a directed path from the *begin* node to the *end* node.

> **Theorem 4:** If $\mathcal{P}$ is a well-connected acyclic graph without conditionals, the *end* node is the last activity in any invocation of $\mathcal{P}$.

The proof is immediate. The firing rule implies that all predecessors of a node must fire before the node fires. In the remainder of the thesis, we assume all acyclic graphs to be well-connected; auxiliary arcs can always be added to ensure this property[15]. In the theory that follows, it is important that graphs be well-connected.

---

[13]The approach developed here is closely related to Leiserson's work on retiming of VLSI circuits [18, 19]. In the retiming work, a node is given a *lead* of 1 when a register is removed from each input arc and a register is added to each output arc. This is essentially like firing a node in a dataflow program. The worst-case token storage corresponds to a worst-case retiming of a circuit.

[14]In practice, instructions with limited side-effects are introduced for special circumstances. These instructions are required to produce an output as well, so it is possible to determine that they have fired.

[15]Note that when auxiliary arcs add added to a graph, they introduce an artificial data dependency. The destination operation is not enabled until data arrives on the auxilliary arc, in addition to the other arcs. The data on the auxilliary arc is discarded when the operation executes.

5.2. Constraint Systems to Model Program Configurations

To motivate the approach, consider an acyclic code-block in execution. Initially, a single token is available at the input node (*i.e.*, the *begin* operator). At each step, some number of enabled activities fire according to the following rules: (i) an operator may fire only if it has tokens available on all its input arcs, and (ii) upon firing, it removes a token from each input arc and produces one on each output arc. After the *Begin* operator fires, the tokens in the graph form a wavefront which partitions the graph into two components: 'fired' nodes and 'unfired' nodes. Whenever a node fires, the wavefront advances and the node moves from the 'unfired' set to the 'fired' set. Note that there are generally many legal execution orders; these correspond to the different ways the wavefront can advance. In any legal configuration, the arcs carrying tokens for a *cut*, which partitions the graph into two disjoint subsets.

The observations above can be expressed concisely in algebraic terms. Let the nodes of the code-block be represented by the set $V = \{v_0,...,v_{n+1}\}$, where v_0 is the *Begin* node and v_{n+1} is the *End* node. The source node, v_0, provides all input tokens. The sink node, v_{n+1}, receives all results. Since graphs are assumed to be well-connected, every node is on a directed path from the source to the sink. In the initial configuration, v_0 has fired once and there is a single token on each of its output arcs.

Consider an arbitrary legal configuration $\mathcal{C}$ of an acyclic code-block. We associate with $\mathcal{C}$ a *firing function* $f_{\mathcal{C}} : V \rightarrow \mathcal{N}$, such that $f_{\mathcal{C}}(i)$ denotes the number of times that v_i has fired in a legal execution order producing $\mathcal{C}$. Note that this is well-defined for acyclic blocks without conditionals, since all execution orders producing a given configuration generate the same firing function. By definition $f_{\mathcal{C}}(0) = 1$. Since the code-block is acyclic, each node fires at most once; thus $f_{\mathcal{C}}(i) \in \{0,1\}$ for all $v_i \in V$. Suppose a node v_i has predecessors $v_{j_1}, v_{j_2},..., v_{j_p}$. The firing rule implies that a node can fire if and only if all its predecessors have fired. Thus, if all of its predecessors fire in an execution order producing $\mathcal{C}$, then v_i may fire or not fire. If any predecessor does not fire, then v_i can not fire. Hence, if $f_{\mathcal{C}}(j_r) = 1$, for $r = 1$ to p, then $f_{\mathcal{C}}(i)$ may be 0 or 1, but if some $f_{\mathcal{C}}(j_r) = 0$ then $f_{\mathcal{C}}(i) = 0$. In either case, $f(i) \leq f(j_r)$, for $r = 1$ to p. This gives the following lemma.

Lemma 5: For any legal configuration $\mathcal{C}$, the corresponding firing function $f_{\mathcal{C}}$ satisfies:

$$f_{\mathcal{C}}(j) \leq \mathrm{MIN}\,\{ f_{\mathcal{C}}(i) : (v_i,v_j) \in E \}, \text{ for all } v_j \in V. \tag{1}$$

We say $f: V \rightarrow \mathcal{N}$ is a *legal firing function* if there exists a legal configuration $\mathcal{C}$ such that $f = f_{\mathcal{C}}$. Lemma 5 yields a set of constraints, one for each edge in E, which are satisfied by any legal firing function:

$$\begin{aligned} &f(0) = 1 \\ &f(n+1) = 0 \\ &f(j) - f(i) \leq 0, \text{ for all } (v_i,v_j) \in E \\ &f(i) \text{ integer, for all } v_i \in V \end{aligned} \tag{2}$$

The space of functions satisfying (2) includes all legal firing functions. Thus, the optimum of the feasible region for the constraints in (2) provides an upper bound on the resource requirements of legal configurations. Moreover, by demonstrating the converse of Lemma 5, we can show that such a bound is tight in that some configuration actually achieves the bound.

Lemma 6: Let $\mathcal{G} = (V, E)$ be the graph of an acyclic code-block without conditionals and $f: V \rightarrow \mathcal{N}$ a function satisfying (2). Then there exists a unique legal configuration $\mathcal{C}$ of $\mathcal{G}$ such that $f_{\mathcal{C}}(i) = f(i)$, for all $v_i \in V$.

Proof: Since every node is on a path from v_0 to v_{n+1}, we have $0 \leq f(i) \leq 1$, for all v_i in V. f is restricted to be integer, so its range is {0,1}. Consider any node v_i such that $f(i) = 1$. Let P be an arbitrary path from v_0 to v_i. Then, for all v_j on P, we have $1 = f(0) \geq f(j) \geq f(i) = 1$. Therefore, the set of nodes which are assigned a value of 1 by f form a connected subgraph which includes v_0. The configuration corresponding to f has one token on each arc (v_i, v_j) such that $f(i) = 1$ and $f(j) = 0$. Any topological ordering on the 'fired' subgraph gives a legal execution order which generates this configuration.□

5.3. Token Storage Requirements of Acyclic Code-blocks

Since legal firing functions correspond directly with legal configurations, the constraint system in (2) provides the first step in determining the worst-case token storage requirement over all possible legal configurations. The set of legal configurations is precisely defined by the space of functions which satisfy (2). The next step is to determine the number of tokens present in the configuration corresponding to a given legal firing function.

Consider an edge (v_i, v_j) of a dataflow graph. Every time v_i fires, a token is produced on this edge. Every time v_j fires, a token is removed from this edge. Therefore, the number of tokens on (v_i, v_j) in a configuration $\mathcal{C}$ is the difference in the number of firings, $f_{\mathcal{C}}(i) - f_{\mathcal{C}}(j)$. Thus, the token storage requirement for the configuration with firing function f is given by

$$TS(f) = \sum_{(i,j) \in E} f(i) - f(j)$$

or, equivalently,

$$TS(f) = \sum_{v_i \in V} f(i) \cdot (\text{Outdegree}(v_i) - \text{Indegree}(v_i))$$

Therefore, the worst-case token storage requirement over all possible legal configurations is obtained by maximizing TS (f), subject to (2).

Note that the constraints $f(0) = 1$ and $f(n+1) = 0$ can be replaced by $f(0) - f(n+1) \leq 1$, since the cost function is unaffected if all the $f(i)$ are scaled by a constant factor. Let c_i denote the quantity Outdegree(v_i) - Indegree(v_i). Linear program 7, below, gives the token-storage requirement of an acyclic block $\mathcal{P} = (V,E)$ without conditionals.

Linear program 7: Token Storage Requirement of Acyclic Code-Block $\mathcal{P} = (V,E)$

Maximize $\Sigma\, c_i f(i)$, subject to

$f(j) - f(i) \leq 0$, for each $(v_i, v_j) \in E$,

$f(0) - f(n+1) \leq 1$, and

$f(i)$ integer, for all $v_i \in V$.

Linear program 7 is the dual of a min-cost flow problem [13]. Since polynomial time algorithms exist to solve min-cost flow problem [20], this linear program can be solved in polynomial time.

More generally, the constraint matrix in the integer linear program derived for an acyclic block without conditionals is *totally unimodular*[16]. This allows the integrality constraint to be ignored, because for a linear program with a totally unimodular constraint matrix and integral right-hand side, every basic feasible solution is integral. We call the linear program obtained from an integer

[16]A matrix is totally unimodular if every non-singular submatrix has determinate 1 or -1. Sufficient conditions for total unimodularity can be found in [20], page 317.

linear program by dropping the integrality constraint the *relaxed linear program*. For acyclic blocks without conditionals, it is sufficient to solve the relaxed linear program, since an optimal solution is integral. The relaxed linear program can be solved by standard methods, *e.g.*, simplex method or network simplex method, or in polynomial time using the Ellipsoid algorithm.

As a simple example of this technique, consider the graph in Figure 5-1. The linear program tableau for this graph is given in Figure 5-2. The constraint matrix is simply the incidence matrix for the graph with an additional edge from v_{n+1} to v_0, except the signs are reversed. The optimal solution has a cost of 3 with $f_0 = f_1 = 1$ and $f_2 = f_3 = 0$.

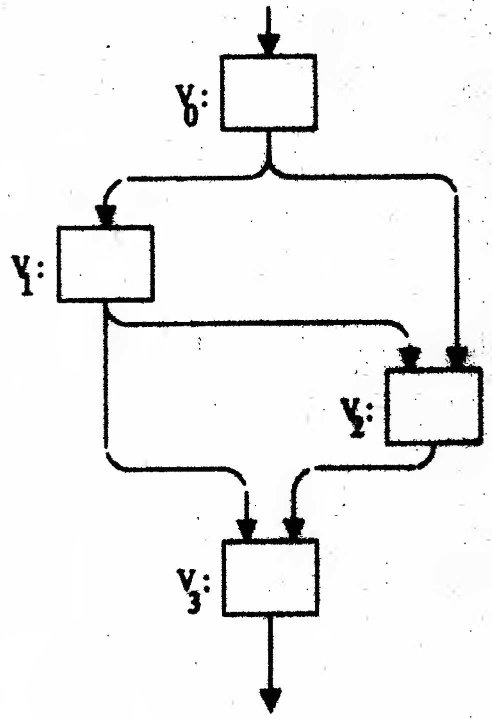

Figure 5-1: Basic Acyclic Block

It was noted above that Linear Program 7 is the dual of a min-cost flow problem. The dual problem has a very simple structure, so it is likely that a fast algorithm could be specially tailored for it. The edges in $\mathcal{P}$ have cost zero and the auxiliary edge from v_{n+1} to v_0 has cost 1. The supply at node v_i is equal to indegree(v_i) - outdegree(v_i), in the original graph. The objective is to minimize the flow along the auxiliary arc. The network corresponding to the example above is shown in Figure 5-3

nodes	$f(0)$	$f(1)$	$f(2)$	$f(3)$		
costs	2	1	-1	-2		
edge e_1	-1	1			$\leq$	0
e_2	-1		1		$\leq$	0
e_3		-1	1		$\leq$	0
e_4		-1		1	$\leq$	0
e_5			-1	1	$\leq$	0
e_6	1			-1	$\leq$	1

Figure 5-2: Example Tableau for Token-Storage Requirement

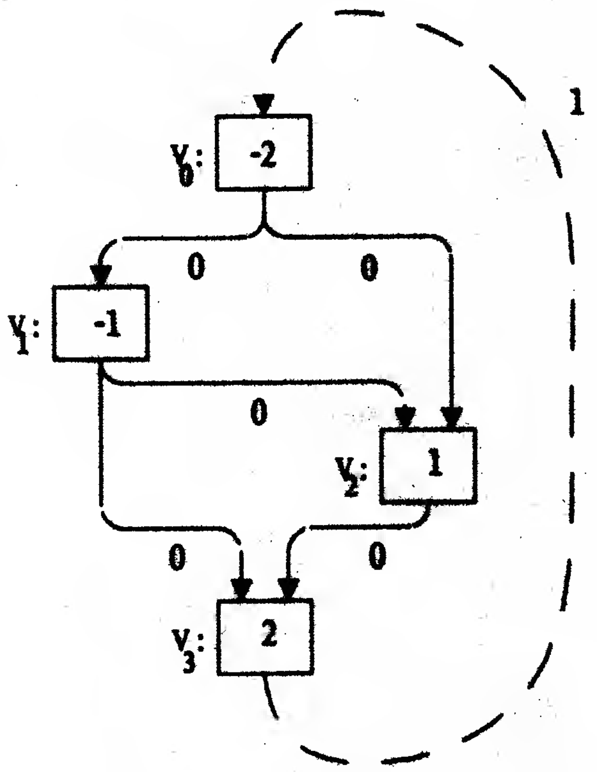

Figure 5-3: Example Min-Cost Flow Problem

It should be noted that the restrictions implied by the dataflow firing rules are crucial to the approach presented here. A legal configuration determines a cut which seperates v_0 and v_{n+1}. Thus the maximum cut of the corresponding undirected graph is certainly an upper bound on the token storage requirements, in the acyclic case. However, determining the max cut of a graph is NP-complete, in general [16]. The key observation is that legal configurations correspond to a restricted class of cuts, ***directed cuts***, which have the property that edges only cross the cut is one direction. In graph theoretic terms, the result above demonstrates that the maximal directed cut of an acyclic graph can be determined in polynomial time. Directed cuts are less interesting in cyclic graphs, because no cycle can cross a directed cut. The generalization employed in network flow problems considers only the edges which cross the cut in the forward direction. However, this generalization is of little value for the restricted class of cyclic graphs permitted in dataflow programs; thus we do not pursue it further.

5.4. Extensions for Multiple Token Stores

In Chapter 3 we suggested that the Tagged Token Dataflow Architecture should be augmented to include (i) a token buffer in the input section for tokens which do not require partners, and (ii) a facility for buffering matched pairs within the wiating matching section. With this change to the machine, there are really two forms of tokens. The load on the two token stores should be accounted for separately.

To extend the basic technique to handle this situation, weights are associated with the edges. To compute the load on the waiting-matching store, we assign a weight of 1 to the input arcs of dyadic operations and a weight of 0 to the rest. To compute the load on the input section token buffer, inputs to monadic operations are assigned a weight of 1 and the rest 0.

Let $\mathcal{P} = (V,E,W)$ be an acyclic code-block with ***token weights*** associated with the edges. Thus, w_{ij} is the maximum number of tokens implied by a token on edge (v_i, v_j). The cost of a legal firing function f is given by:

$$\mathrm{WTS}(f) = \sum_{(i,j) \in E} (f(i) - f(j)) \cdot w_{ij} \tag{3}$$

or, equivalently,

$$\mathrm{WTS}(f) = \sum_{v_i \in V} f(i) \cdot (\mathrm{Outweight}(v_i) - \mathrm{Inweight}(v_i)),$$

where Outweight(v) and Inweight(v) are the sum of the weights on arcs emanating from and incident to node v, respectively.

Thus, for each code-block we solve two linear programs, one for each form of token.

5.5. Token Storage on Individual PEs

In the Tagged Token Dataflow Architecture, when an acyclic code-block is invoked it is spread over the PEs in a domain. Each PE provides storage for certain tokens generated during the course of the invocation. Suppose a code-block with a token storage requirement of *t* tokens is spread over a collection of PEs, what is the worst-case token storage requirement experienced by an individual PE? It may be as large as *t* tokens. However, usually the worst-case requirement for an individual PE will be much less than *t*. A PE must provide storage only for tokens on arcs which provide input to instructions it contains. Given a partition on the code-block, we want to determine the worst-case token storage for each PE. The edge-weighting techniques introduced above can be employed to handle this situation.

Let $V = (V_1, \dots, V_l)$ be a partition on a code-block into *l* components. The cost function for partition V_m is given by (3), where

$$w_{i,j} = \begin{cases} 0, & \text{if } v_j \notin V_m \\ 1, & \text{if } v_j \in V_m \end{cases}$$

5.6. Resource Requirements of Entire Acyclic Programs

The edge-weighting technique can also be applied to determine the worst-case token storage, CBR, or constant area requirement of an entire invocation tree. Suppose code-block $\mathcal{P}$ invokes code-block $\mathcal{Q}$ and the worst-case token storage requirement of the subtree rooted at $\mathcal{Q}$ is T. Consider the apply schema in the graph of $\mathcal{P}$ where $\mathcal{Q}$ is invoked. A single token on the arc between the *apply* and *apply*$^{-1}$ stands for as many as T tokens in the subtree initiated by this invocation. Therefore, in determining the worst-case token storage requirements for the entire subtree rooted at $\mathcal{P}$, a cost of *T* should be associated with a single token on this edge.

The worst-case token storage requirements of a tree of non-recursive applications can be determined by working up from the leaves. The code-blocks which appear as leaves can be solved by the basic method (*i.e.*, all edge weights are unity). The resulting token storage requirements are used as the weights on edges where these code-blocks are invoked. The requirements of a code-block can be determined as soon as all its immediate subordinate blocks are determined. Note that even though a given code-block may appear many times in the overall tree, it need be solved only once. The algorithm is given below.

Algorithm 8: Token Storage Requirement of an Entire Program

Input. Program $\mathcal{P}$ comprised of acyclic code-blocks without conditionals, $C_1,\ldots, C_k$.

Output. Worst-case token storage requirement of $\mathcal{P}$.

Method.

1. Construct a directed graph $\mathcal{I}$ of invocation relationships: $(C_i, C_j) \in \mathcal{I}$ if C_i invokes C_j. If $\mathcal{I}$ has a cycle, stop; the program is recursive and can not terminate.
2. Construct a topological ordering $O = C_{j_1}, \ldots, C_{j_k}$, such that C_j precedes C_i in O if there is a path from C_i to C_j in $\mathcal{I}$.
3. Solve the edge-weighted token storage linear program for each code-block in the order given by O, using the results of $C_1,\ldots,C_{i-1}$ as weights in the program for C_i. Return the result for the top-level code-block C_k.

Algorithm 8 can be applied to determine the worst-case CBR requirement of such a program as well. Edges between *apply* and an *apply*$^{-1}$ operation should be weighted by the CBR requirement of the subtree generated by the *apply*. All other edges receive a weight of zero. The CBR requirement of the subtree rooted at a given code-block is 1 plus the result of the edge-weighted linear program.

Constant area requirements can be determined in a similar fashion. Edges between *apply* and an *apply*$^{-1}$ operation should be weighted by the constant area requirement of the subtree generated by the *apply*. All other edges receive a weight of zero. The constant area requirement of the subtree rooted at a given code-block is the constant area size for that code-block plus the result of the edge-weighted linear program.

5.7. Summary

This chapter has made significant contributions towards solving our resource management problems for a restricted class of programs. Let us review the contributions so far.

1. *Termination Detection:* Solved. By adding arcs so that every node is on a path from the *begin* to the *end*, the firing of the *end* signifies the termination of an invocation. This does not say *a priori* whether a program will or will not terminate, but whenever a code-block invocation terminates the resource manager will be informed.

2. *Token Storage Overflow*: Solved. A linear programming technique can be used to determine the worst-case token storage requirement of a code-block activation. This allows the resource manager to avoid over-committing any token storage unit. This does not imply the program will run to completion, but it will not halt because of token storage overflow.

3. *Iteration Overflow*: Non-issue with acyclic code-blocks.

4. *Program Deadlock*: Partially addressed. The linear program technique can be extended to determine the worst-case resource requirements of overall programs, for this restricted class. For a given program, if the resource bound derived in this way is less than the capacity of the machine, the program will execute to completion. If the resource bound exceeds the capacity of the machine, the program may or may not execute to completion, depending on the execution order that is pursued. We have no addressed how to bias the execution order to reduce the resource requirements, but we will return to this topic.

The major stumbling block in analyzing acyclic graphs without conditionals is the indeterminacy in the execution order. We are forced to consider all possible execution orders, since the particular order that will be followed by a specific machine is impossible to predict. This stumbling block is overcome by representing the space of legal configurations as a system of integer linear constraints. This chapter demonstrates the power of this approach through a variety of applications involving token storage, CBR, and constant area requirements.

Chapter Six

Analysis of Conditional Blocks

As a second step in addressing the resource management problems, we consider acyclic blocks with conditionals. The introduction of conditionals changes the complexity of program analysis dramatically. With code-block invocation and conditional expressions, the model is fully general, *i.e.*, we can express all computable functions as dataflow programs of this form. Determining the resource requirements for entire programs is equivalent to solving the classic halting problem, since for acyclic graphs bounded resource requirements imply termination. Thus, we can not hope to extend all the results in Chapter 5 to handle this more general class of programs. The results concerning *entire* programs hold only if recursion is excluded. The results concerning individual code-block invocations hold, but are more difficult to compute. A weak upper bound on the token storage requirement of an acyclic block with conditionals is the number of arcs in the graph. The question is how tight a bound can we achieve with reasonable effort. Determining absolutely tight bounds proves to be quite difficult. We can not easily eliminate conditionals to reduce the problem to the case handled in Chapter 5. We can derive a constraint system to model the special behavior of conditionals, but unfortunately these constraints do not exhibit the special form that was exploited in Chapter 5. Indeed, we can not hope to solve the resulting integer linear programs efficiently in all cases, because we can show that determining tight bounds is NP-complete. Thus, we are faced with developing algorithms which give good bounds for the programs encountered in practice. A simple branch-and-bound technique provides such an algorithm.

6.1. Termination Detection

Determining when all activities for an invocation have completed is complicated by the presence on conditionals. We must guarantee that the *end* node is the last to fire, under any setting of the conditionals. It is reasonable for a *switch* to have no output arc on one side or the other, since a sub-block of a conditional may not require all the inputs. In this circumstance, it is difficult to determine whether the *switch* has fired.

We say a conditional is *well-connected* if (1) every *switch* is on a path to a *merge* under either setting of the conditional, and (2) every node in the conditional is on a path from a *switch* to a *merge* Note, a *switch* need not be connect to the same *merge* under both settings of the conditional.

Theorem 4 holds for well-connected conditionals. In order for all the *merges* to fire, the entire conditional must be complete. No tokens remain in either sub-block. There are no more activities to fire. For any well-connected acyclic graph of well-behaved operators and well-connected conditionals, the *end* node is the last activity. We assume, in the remainder of the thesis, that all conditionals are well-connected. Auxiliary arcs can be added so this condition is met.

Note, well-connectedness does not imply that a given code-block invocation will terminate. It only implies that when an invocation does terminate, the event can be detected.

6.2. Naive Approaches to Token Storage Analysis

The basic conditional schema was discussed in Chapter 2. One might hope to analyze the two sub-blocks of the conditional in isolation and replace the entire conditional with a simpler structure which would provide the same worst case. This approach worked well with subordinate code-block invocations in Chapter 5. Unfortunately, it fails for conditionals. The essential factor is the strictness of the block being analyzed. Code-blocks are strict; they do not execute until all their inputs are available and do not complete until all their results are produced. This property is enforced by the restriction that a single argument token initiates an invocation and a single result is produced. The sub-blocks within a conditional are not strict. Thus, their internal structure plays an important role.

As an example, consider the graph in Figure 6-1. It has a large fan-in above the righthand switch and a large fan-out below the righthand merge. A 'false' setting (shown as bold arcs) allows both the large fan-in and the large fan-out to contribute to the token storage requirement. Tokens can pass through the left-hand switch and enter the large fan out below the right-hand merge, even if tokens for the right-hand switch get stuck in the area of large fan-in. A 'true' setting does not allow the fan-in and fan-out to both contribute to the token storage requirement. Tokens must drain out of the region of large fan-in and enable the right-hand switch, before tokens can enter the region of large fan-out. The two sub-blocks have equal storage requirements in isolation, but either could be made to have arbitrarily large storage requirements without changing the essential dependencies. Note that for conditionals with a single input and a single output, it is valid to replace the conditional with a single arc, weighted by storage requirements of the worst sub-block.

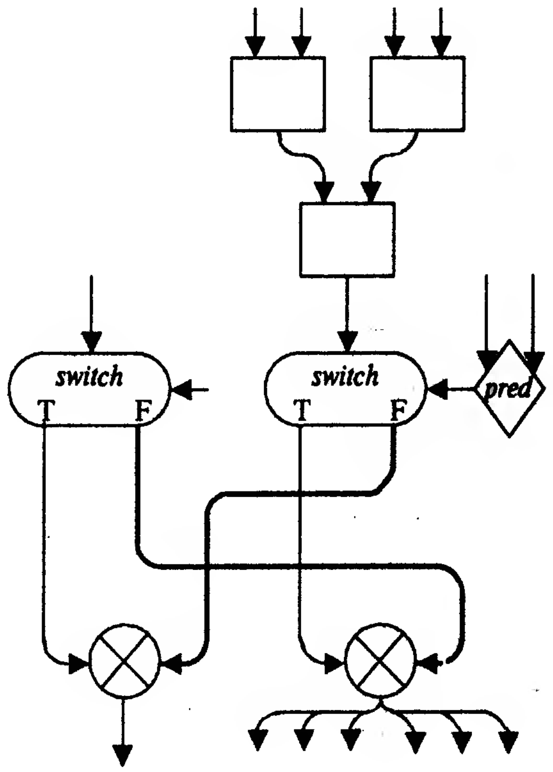

Figure 6-1: Sub-Block Replacement Counter Example

This example also demonstrates that is it not valid to ignore the special behavior of the *switch* and *merge* and pretend that both sub-blocks receive tokens. The merge would have to be treated as a strict operator (*i.e.*, requiring tokens on all its input arcs), and this would prohibit the large fan-in and large fan-out from both participating in the worst-case storage requirement.

6.3. Constraint Model for Conditionals

The approach employed in Chapter 5 can be extended to model the execution of conditional blocks. New kinds of constraints must be introduced to capture the special behavior of the *switch* and *merge* operators. A *switch* consumes a token from each of its input arcs, but produces a token only on one or the other of its output arcs. Suppose *switch* s_j provides input to t_j and f_j, then

$$f(t_j) + f(f_j) \leq f(s_j).$$

No special treatment is required for the input arcs[17].

A *merge* fires whenever a token is present on either input arc. It consumes a token from only one input arc, and produces a token on each output arc. Suppose merge m_i receives input from t'_i and f'_i, then

$$f(m_i) \leq f(t'_i) + f(f'_i).$$

No special treatment is required for the output arcs.

Additional constraints are required to capture the fact that all the switches route data to the same sub-block. Thus, if the input nodes of the 'true' sub-block are $t_1, \ldots t_k$, and the input nodes of the 'false' sub-block are $f_1, \ldots, f_k$, we have

$$f(t_i) + f(f_j) \leq 1, \text{ for all } i,j = 1 \text{ to } k.$$

This ensures that only one side of the conditional executes.

The tokens on the outputs of a switch must be accounted for collectively in the objective function; the number of tokens on the outputs of switch s_i is given by $f(s_i) - f(t_i) - f(f_i)$. The tokens on the inputs to a merge are given by $f(m_i) - f(t'_i) - f(f'_i)$.

The constraint system generated in this manner gives an integer linear program whose optimal solution is a tight bound on the token storage requirements of an acyclic code-block with conditionals. This is only a first step, because in general integer linear programming is NP-complete. To be practical, an efficient technique must be developed for the particular class of integer linear program at hand. In Chapter 5 we noted that the constraint matrix generated for

[17]We assume that *switch* operators have a single 'true' output arc and a single 'false' output arc. This makes the presentation much simpler, but is not necessary in practice.

acyclic blocks without conditionals is totally unimodular. As a result, the integrality constraint can be ignored, and the relaxed linear program solved using standard techniques. Unfortunately, the constraint systems for acyclic blocks *with* conditionals do not exhibit this simple structure. A sufficient condition for total unimodularity is that a 1 and a -1 appear in every row, as the only nonzero entries. The constraint matrix derived for conditionals does not have this property, because the *switch* and *merge* introduce rows with three non-zero entries.

Of course, failing to meet this sufficient condition does not prove that the relaxed linear program for an acyclic block with conditionals will fail to have an integer optimal solution. However, the result proved in the next section essentially proves this to be so. The optimal solution of the relaxed linear program gives an upper bound on the storage requirements. If the optimal solution for a given program happens to be integral, the bound is tight since the optimal solution represents a legal firing function. The relaxed linear program can be solved in polynomial time. Thus, if for every acyclic blocks with conditionals, the relaxed linear program has an optimal integer solution, tight bounds for acyclic blocks with conditionals can be determined in polynomial time. However, the next result shows that determining tight bounds for such graphs is NP-complete.

6.4. NP-Completeness of Tight Storage Bounds

The problem of finding a tight upper-bound on the maximal storage requirements of an acyclic block with conditionals is NP-complete.

Definition 9: Let MAXCOND = { $\langle \mathcal{P}, k \rangle$: $\mathcal{P}$ is an acyclic code-block with conditionals which has a storage requirement of at least k on some legal execution sequence. }

Theorem 10: MAXCOND is NP-complete.

Proof: The proof is a reduction from satisfiability of boolean formulas in conjunctive normal form (SAT). MAXCOND is in NP because we can guess the arcs which contain tokens in the worst case and verify this (i) includes k arcs, and (ii) is a legal configuration. To check that the configuration is legal, guess the setting of the control variables and simulate the execution, without allowing any tokens to be removed from the chosen arcs.

For the completeness, suppose φ is an instance of SAT with clauses $\langle C_1, ..., C_k \rangle$ and variables $x_1, ..., x_n$. Construct an acyclic graph with n conditionals in sequence, each k inputs wide (cf. Figure 6-2). The data inputs denote the clauses. The control inputs denote the variables. If x_i appears in C_j, the true side of the corresponding conditional is given a sub-block with a cut of two; otherwise, it is given a cut of one. Similarly, the false side has a cut of two, if x_i appears complemented in C_j, and one, otherwise. In any legal configuration, each C_j path will contain one or two tokens, and if there are two they must be in the same sub-block. φ is satisfiable if and only if the graph has a maximum storage requirement of 2k. □

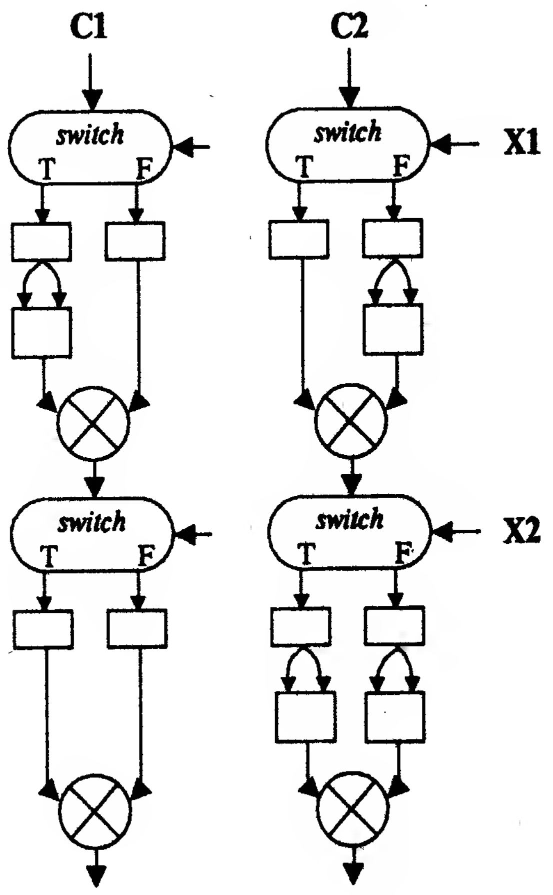

Figure 6-2: Graph for $(X_1) \wedge (\neg X_1 \vee X_2 \vee \neg X_2)$

6.5. Approximate Bounds

Since determining tight bounds on blocks with conditionals is inherently complex, we should investigate how good bounds can be derived for the graphs encountered in practice. Two rather extreme approaches suggest themselves. (1) Given an integer linear program for an acyclic code-block with conditionals, the solution to the relaxed linear program offers an upper bound on the storage requirements. This relaxed program can be solved efficiently. The quality of the bound for a given code-block depends on the particular structure of the code-block. (2) Tight bounds can be derived by brute force. Given a code-block with *k* conditionals, eliminate one conditional by formulating two sub-problems, one assuming the 'true' side is enabled and one assuming the 'false' side. Solve both recursively. This provides a tight bound for any code-block, but unfortunately the running time is exponential in the number of conditionals.

These two approaches can be combined to give a successive refinement approach, employing branch-and-bound techniques. Formulate and solve the relaxed program to get an initial upper bound. If the solution is integral, stop; the bound is tight. This initial bound is an upper bound on the storage requirements, regardless of which way the conditionals fire. The bound can be successively improved by further constraining the problem. Choose a conditional which is partially determined (*i.e.*, for which both arms are partially fired). Generate two sub-problems by forcing one arm not to fire and then forcing the other. This can be accomplished by adding constraints of the form $f(t_i) = 0$, and then $f(f_i) = 0$. Note, however, that this new system of constraints can be simplified, eliminating the constraints from the sub-block which was forced not to fire and putting the constraints for the *switch* and *merge* into the usual, simple form. The solutions to these two new problems represent improved bounds on the storage requirements of the block, with an assumption about how a particular conditional fires. The maximum of the two (both are guaranteed to be no larger than the bound given by the parent problem) can be taken as a bound on the storage requirement of the code-block.

This process can be continued, generating a tree of sub-problems. At any point, the maximum value at the leaves represents the best approximation of the worst-case storage requirement. The important point is that the entire tree need not be developed. At any point, the process may be terminated and the partial tree offers an upper bound on the storage requirement. As the tree is extended the bound is improved. Note that the value determined for a particular sub-problem is always as large as the values determined for either of its children. Thus, a branch-and-bound approach can be employed to avoid expanding parts of the tree. A node can be eliminated if the value determined at some other leaf node (of the partial tree) is greater. The algorithm is given below.

Algorithm 11: Token Storage for Acyclic Blocks with Conditionals

Input. Acyclic code-block $\mathcal{P}$ with conditionals.

Output. Approximate token storage requirement of $\mathcal{P}$.

Method.

1. Generate and solve the relaxed linear program for $\mathcal{P}$.
2. Repeat until either an integral solution is obtained or some prespecified number of refinements have been attempted.
    a. Select the leaf node v of the partial solution tree with the greatest value.
    b. If v has an integral optimal solution, stop; no other setting of the conditionals can generate a larger value and there exists a legal execution which meets the stated bound.
    c. Otherwise, expand v by forcing a partially determined conditional and simplifying the constraint matrix. Solve both subproblems.

Using this branch-and-bound technique, we can solve (to a reasonable approximation) the various constraint systems developed in Chapter 5. Weighting the edges in the graph simply changes the coefficients in the objective function. Algorithm 11 can still be employed. The resource requirements of entire programs can only be determined if recursion is excluded.

6.6. Summary

We have now extended our treatment of the resource management problems to include a fully general class of programs. Let us recapitulate the contributions so far.

1. *Termination Detection*: Solved. By adding arcs to graphs, it is possible to guarantee that the firing of the *end* node always signifies termination of a code-block invocation.

2. *Token Storage Overflow*: Solved, but conservatively. By employing the linear programming techniques in a successive refinement algorithm we can generate a reasonably tight bound on the token storage requirements of individual code-block invocations. Reserving this much storage to an invocation will guarantee that overflow is avoided, but may be slightly wasteful, since it is possible that no legal configuration actually achieves the bound.

3. *Iteration Overflow*: Non-issue with acyclic graphs.

4. *Program Deadlock*: Not addressed. The results in Chapter 5 break down in the face of recursion, because we can not determine in general whether a program terminates. We can not determine the depth of a branch of the invocation tree, *a priori*, thus we can not determine whether a program will exhaust the machine resources before completing.

Acyclic graphs with conditionals present two problems: indeterminacy in the execution order, and data dependent behavior. The former can be addressed with constraint system techniques, but the latter makes many problems that can be solved efficiently without conditionals NP-complete, when conditionals are included. For most dataflow programs encountered in practice, the complexity can be largely overcome by a method of successive refinement.

Chapter Seven

Analysis and Control of Loops

The resource management problems are particularly serious when loop code-blocks are involved. The token storage requirement of a loop may be extremely large and dynamic, since it depends on the number of iterations executing concurrently. Iteration identifiers should be recycled automatically as a loop progresses. Loop code-blocks may generate many subordinate invocations, causing the active portion of the invocation tree to grow very broad. Rapidly spawning off many large, independent computations makes program deadlock a likely occurrence. The latter three resource management problems are closely related; they boil down to a single issue: how to control the unfolding of loops so the number of concurrent iterations can be bounded.

We begin our study of loops by extending the constraint technique to model the execution of loop code-blocks. The problems encountered with *switch* and *merge* nodes in Chapter 6 can be overcome rather easily. The resulting linear program can be solved efficiently, but may not have a bounded optimal solution. This suggests an important classification of loops: some loops have bounded resource requirements, others do not. This classification is essential, for a loop has bounded resource requirements if and only if it has bounded unfolding, *i.e.*, if it can generate at most a bounded number of concurrent iterations. A structural characterization can be given for the two classes of loops: a loop has bounded resource requirements (and bounded unfolding) if and only if the loop body forms a single, strongly connected component. This structural charaterization suggests how unbounded loops can be transformed into bounded loops with the addition of minimal number of dependency arcs. These transformed loops exhibit controlled unfolding. In fact, the maximum unfolding can be adjusted dynamically. They have precisely the properties we desire from a resource management viewpoint. Tokens storage requirements can be determined in advance. A fixed collection of iteration numbers can will be recycled automatically. The number of concurrent subordinate invocations can be controlled.

7.1. Termination Detection

Determining when loop code-block invocations are complete is somewhat more subtle than for acyclic code-blocks. We must guarantee that all iterations have completed. To this end, the approach adopted in the preceding chapters can be extended to ensure that the *end* node fires only when the invocation is complete.

We say a loop code-block is *well-connected* if:

1. all conditionals it contains are well-connected,

2. all nodes in the header are on a path from the *begin* node to a *merge*,

3. all nodes in the cyclic portion are on a path from a *merge* to a *D* operator,

4. the 'true' output of every *switch* is on a path to a *D* operator,

5. the 'false' output of every *switch* is connected to a D^{-1} operator, and

6. all nodes in the trailer are on a path from a D^{-1} operator to the *end* node.

Condition 1 is necessary to ensure that the header, body, predicate, and trailer behave as well-connected acyclic blocks. Condition 2 ensures that if every *merge* has received an input, the header has competed. Conditions 3 and 4 ensure that if every *D* operator has received a token for iteration i, then iteration i has completed. By induction, all iterations previous to i have completed as well. Condition 5 ensures that if every D^{-1} operator has received input, then all iteration of the loop are complete. Finally, condition 6 ensures that the invocation is complete when the *end* node fires.

7.2. Constraint Model of Loop Configurations

The basic loop schema is shown in Figure 2-6, in Chapter 2. Recall, the header, body, and trailer are acyclic blocks. The outputs of the header form the initial inputs to the *merges*. Tokens circulate through the loop body until the loop predicate turns false. The final wave of tokens is routed to the trailer block. Every cycle in the graph is broken by exactly one op(merge), one *switch* operator, and one D operator, For the purposes of this discussion, the deterministic merge will be regarded as a true operator, even though it can be implemented by allowing two arcs to converge on the same port. In developing the theory for analyzing loops we will assume that no conditional expressions appear within the loop. Toward the end of the chapter we will address the complications introduced by conditionals.

Since we are excluding conditionals within the loop, all operators appearing in a cyclic portion are well-behaved, except the *switches* and *merges* that guide circulating values through the loop body, and hence obey the standard firing rule: when a node fires, a token is consumed from every input arc and one is produced on every output arc. This rule implies that for any legal configuration $\mathbf{C}$, $f_{\mathbf{C}}(i) \geq f_{\mathbf{C}}(j)$, if there is an arc from v_i to v_j. Note that this constraint holds regardless of how many iterations are performed, since every time v_j fires it must consume a token produced by v_i. Therefore, the usual form of constraint holds for any arc between well-behaved operators.

The *switch* and *merge* nodes do not obey this rule; they must be treated specially as in conditional blocks. Again, the constraints hold regardless of the number of iterations. The special constraints which force all the switches to fire the same way are not required for loops. The role of the switch operator in the loop schema is somewhat restricted, because at most one wave of tokens can be routed to the trailer. Thus, if *switch* s_i is connected to f_i on the 'false' side, the constraint $f_{\mathbf{C}}(f_i) \leq 1$ should be included.

Given a loop code-block without conditionals, the set of constraints generated in the manner described above are satisfied in any legal configuration. The analog of Lemma 5.2 holds for loops; any function which satisfies such a set of constraints is the firing function for a legal configuration. However, unlike acyclic blocks, the legal configuration for a given firing function is not unique. Many distinct legal configurations may give rise to the same firing function, since tokens are allowed to become reordered (*i.e.*, the activity for the $i+1^{th}$ iteration of an operator may complete before the activity for the i^{th} iteration). However, the exact values and iteration numbers carried on tokens do not affect the token storage requirements, only the number of tokens on the arcs; thus, all legal configurations for a particular firing function are equivalent in regard to the token storage requirement. The integer linear program for a loop code-block derived from the firing constraints defines the space of all distinct legal configurations of the code-block, with no restriction on the number of iterations that the loop performs.

In dealing with acyclic code-blocks without conditionals, the integrality constraint can be relaxed, with no effect on the optimal solution. The relaxed linear program can be solved with conventional techniques. This simplification is possible because the constraint matrix is totally unimodular; each row of the constraint matrix contains one +1 entry, one -1 entry, and the rest 0. The constraint matrix derived for a loop code-block does not have this property; some rows have three non-zero entries. For loops, the relaxed linear program provides an upper bound on the token storage requirements, but this bound is not necessarily tight. If the optimal solution to the relaxed problem happens to be integral, the bound is tight, however. For a given loop code-block, the extent to which the optimal solution to the relaxed problem differs from the optimal integer solution depends on the particular structure of the graph.

A simpler constraint system can be generated by treating the header, body, and trailer independently. The graph is partitioned into three subgraphs and assumed to have initial conditions as indicated by Figure 7-1. The maximum cut in the header contributes to the token-storage bound, and yet the cyclic portion is treated as if all its inputs were provided. A similar situation holds for the trailer. The *merge* and *switch* operators behave as identity operators. The worst case storage requirements of the three blocks can be determined independently and summed to get a bound on the overall token storage requirements. This bound is slightly loose; it exceeds the actual worst-case storage requirements by a constant factor, not greater than the sum of the requirements of the header and the trailer. However, the simplified constraint systems are totally unimodular, and hence can be solved by conventional methods.

The linear programs for the header and trailer are exactly as in Chapters 5 and 6. The linear program for the cyclic portion is similar, but no auxiliary arc is introduced. The *merges* act as sources and the *D* operators act as sinks. These are paired by the feedback arcs. A single token is assumed to be present on each of these feedback arcs in the initial configuration. In the remainder of the thesis, we ignore the header and trailer portion of loops, focusing on the cyclic portion.

Each *D-Merge* pair is associated with a loop variable, so let the *D* and *Merge* for the i^{th} loop variable be denoted by D_i and M_i, respectively. The constraint for the edge (D_i, M_i) is given by

$$f(M_i) \leq f(D_i) + 1.$$

The number of tokens on arc (D_i, M_i) is $f(D_i) - f(M_i) + 1$, since one arrives from the header. The token storage requirement of the body portion of a code-block are given by the linear program below.

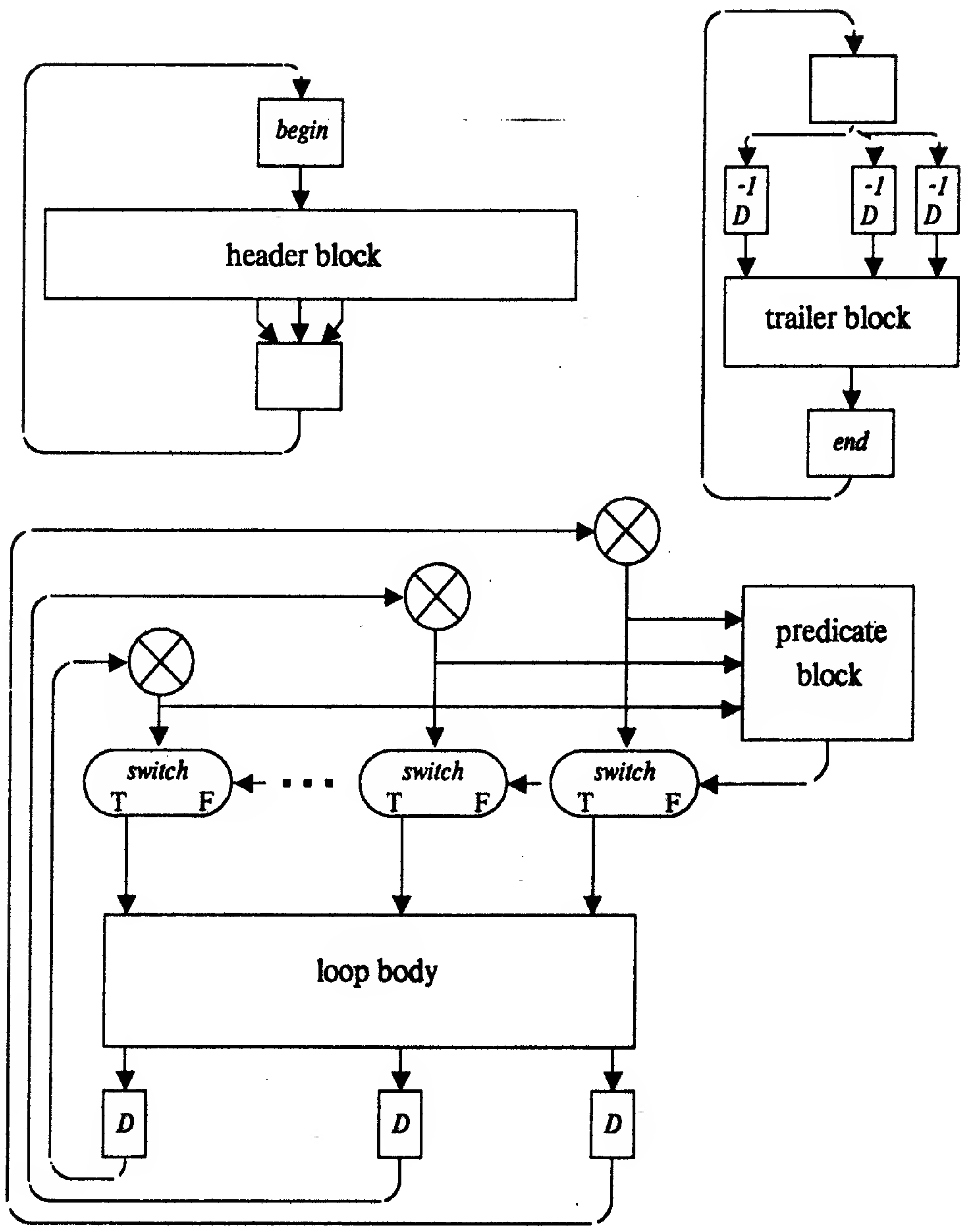

Figure 7-1: Loop Broken Down for Analysis

Linear program 12: Token Storage for Loop Code-blocks.

Maximize $N + \Sigma c_i f(i)$, subject to

$f(v) - f(u) \leq 0$, for all $(u,v) \in E$, v not a merge,

$f(v) - f(u) \leq 1$, for all $(u,v) \in E$, v a merge,

where N is the number of *merge* nodes.

7.3. Bounded and Unbounded Loops

The constraint system above models the legal configurations of a loop, with no restriction on the number of iterations the loop is permitted to execute. Thus, the space of distinct legal configurations defined by these constraints allows for any number of iterations. If the linear program for a loop has an optimal solution, the loop operates within bounded storage requirements, regardless of the number of iterations it performs. In general, this will not be the case. Consider the summation example discussed in Chapter 2. The body portion of the loop is reproduced in Figure 7-2. The index value (I) may circulate through an arbitrary number of iterations without the *switch* for the SUM variable firing even once. Tokens simply accrue on the arcs emanating from the cycle of the index variable. This situation arises because one variable (I) depends only on itself, while the other (SUM) depends on both loop variables. The loop body shown in Figure 7-3 exhibits more controlled behavior. All loop variables are dependent on all other loop variables. Thus, no particular variable can get arbitrarily ahead of the others and cause tokens to pile up. These two examples typify the two basic classes of loop code-blocks: resource unbounded and resource bounded. The class to which a particular loop belongs is entirely determined by the structural properties alluded to in these examples. These ideas are formalized below.

Resource requirements and loop unfolding

The classification of loops can be defined in terms of resource requirements, such as token storage, or in terms behavioral aspects, such as the number of concurrent iterations. The two definitions are equivalent.

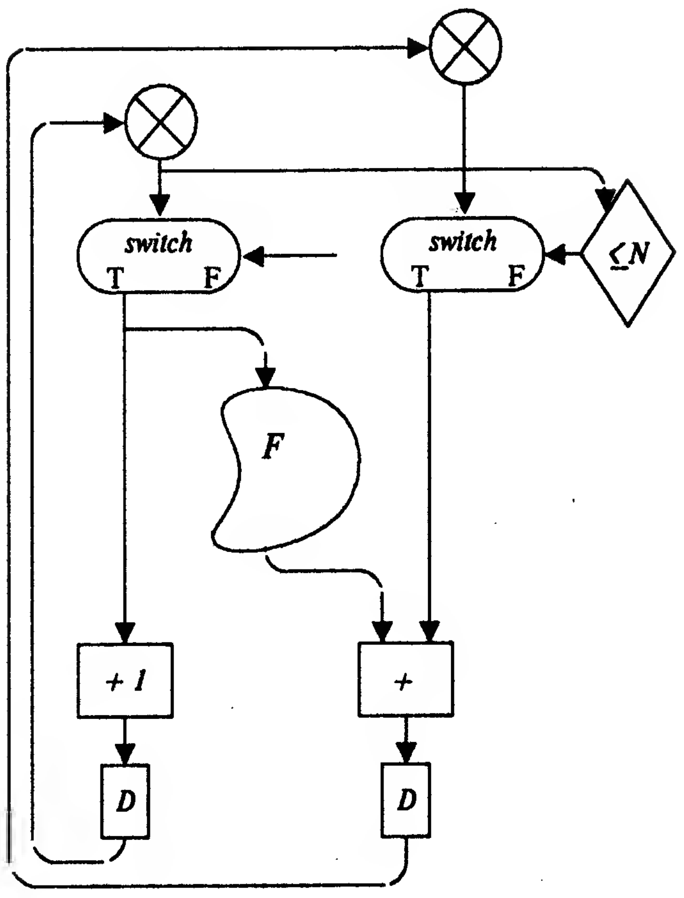

Figure 7-2: Loop Body for Summation

Definition 13: A loop code-block $\mathcal{L}$ is ***storage bounded*** if every legal execution of $\mathcal{L}$ requires at most a bounded amount of token storage, independent of the total number of iterations; otherwise it is ***storage unbounded***.

A loop code-block $\mathcal{L}$ has ***bounded unfolding*** if in every legal execution of $\mathcal{L}$ a bounded number of iterations are active concurrently; otherwise it has ***unbounded unfolding***.

Theorem 14: A loop code-block $\mathcal{L}$ is storage bounded if and only if it has bounded unfolding.

Proof: ($\Rightarrow$) Suppose every legal execution of $\mathcal{L}$ allows at most k concurrent iterations. Let l be such that at any time there are at most l tokens belonging to a given iteration. (The number of arcs suffices for l.) Therefore, there can be at most lk tokens in existence at any time, regardless of the number of iterations executed in total.

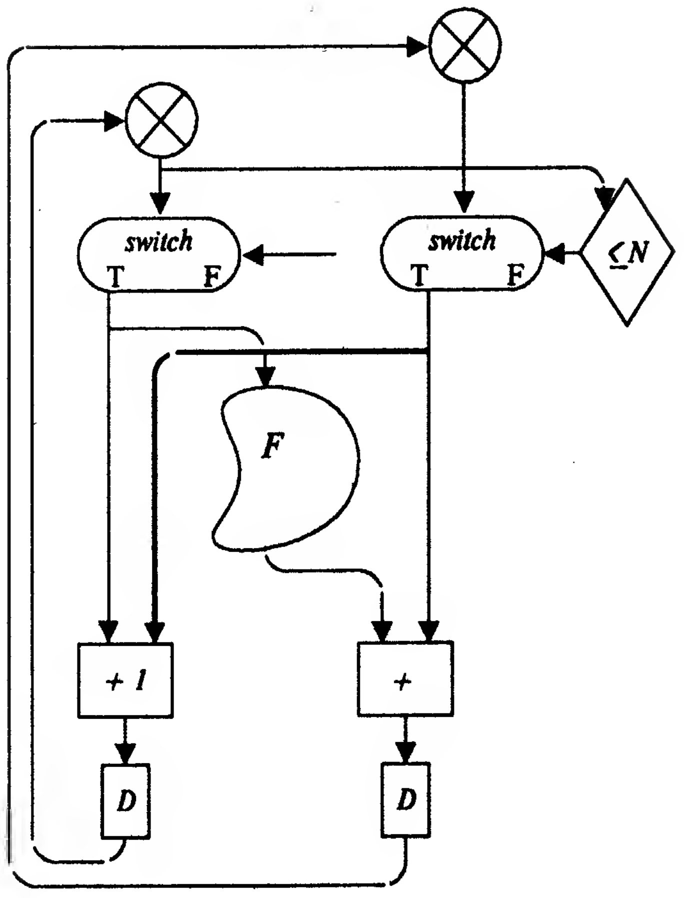

Figure 7-3: A Resource Bounded Loop

(⇐) Suppose that in every legal execution of *L* no more than k tokens are in existence concurrently. Consider first the case where no apply schema appears within the loop. At most k iterations can be in execution concurrently, since at least one token must exist for each active iteration. The apply schema introduces a difficulty because the argument token disappears from the activation and is later replaced by a result token; it is possible that no token exists for an active iteration. Suppose that more than k iterations can be active concurrently. Then all but k of the active iterations must be represented only by subordinate invocations. The firing rule allows all these subordinate invocations to complete (supplying result tokens) before any new activities within *L* fire. In this case, more than k tokens will exists within the invocation, contrary to our assumption.□

This result links together the major resource management problems. If a loop has bounded unfolding then: (1) it can execute within a fixed supply of iteration numbers, (2) the worst-case token storage requirements can be determined in advance, and (3) the branching of the active portion of the invocation tree can be bounded. The first two facts provide direct solutions to token storage and iteration overflow. The latter fact reduces the likelihood of program deadlock. Thus, we should examine how the unfolding of loops can be controlled.

Bounding the number of concurrent iterations only partially solves the iteration overflow problem. There must be an efficient mechanism for recycling iteration identifiers. Ideally, if a *D* operator receives a token with iteration identifier ι, it should produce a token with iteration identifier ι' = ι+1 MOD k, for some k, and be guaranteed that all previous uses of ι' have terminated.

The remainder of the chapter examines bounded and unbounded loops more thoroughly and demonstrates that iteration identifiers can indeed be assigned in the modulo fashion suggested above. In presenting the theory we have need for describing the behavior of programs under the U-interpreter. Thus, when we refer to iteration I of node v, we mean the instance of v that would carry iteration I under the U-interpreter. Once we establish that a certain class of loops make use of a fixed size interval of iteration numbers, it will be clear that the iteration number can be safely replaced by fixed sized iteration identifiers, assigned in modulo fashion.

Structural classification of loops

Under what conditions can a loop code-block potentially unfold into an arbitrarily large number of concurrent iterations? The two example loops above suggest the answer; a loop has bounded unfolding if all loop variables are mutually dependent. Stated somewhat differently, a loop has bounded unfolding if the the body portion form a single ***strongly connected component*** [1].

Definition 15: A node v_j is *k-dependent* on v_i if there exists a path from v_i to v_j which contains k *D* operators, including v_i, but not v_j.

We say v_j is *directly dependent* on v_i, if v_j is 0-dependent on v_i.

We say v_j is *dependent* on v_i, if it is k-dependent for some k.

Lemma 16: If v_j is k-dependent on v_i, then iteration n-k of v_i must fire before iteration n of v_j can fire, for $n \geq k$.

Proof: The result follows directly from the firing rule, with induction on the length of the path. Informally, the token produced by iteration n-k of v_i generates a sequence of tokens which traverse the k-dependency path, yielding an input token to v_j with iteration number *n*. □

Corollary 17: Suppose v_j is k-dependent on v_i. Let e_j be the final arc on the k-dependency path and e_i be an input arc for v_i. Suppose token t_i resides on e_i and t_j on e_j. The iteration number for t_j can not be k larger than that for t_i.

Our goal is to show that for certain loops all pairs of nodes are k-dependent, for sufficiently large k. This motivates the next lemma.

Lemma 18: Let $\mathcal{L}$ be a loop code-block without conditionals. There exists at least one loop variable *i* such that all *D* operators in $\mathcal{L}$ are directly dependent on *merge* M_i. Such an *i* is called a *leading variable.*

Proof: Let *i* be a loop variable such that there exists a directed path from M_i to the loop predicate, which does not pass through any *switch.* There must be such an *i,* since the loop predicate fires once per iteration. Since the loop predicate provides input to every *switch,* each *switch* is directly dependent on M_i. Every *D* is directly dependent on some *switch,* so the result follows.□

Corollary 19: If *i* is a leading variable, every node is 1-dependent upon M_i.

Corollary 20: If *i* is a leading variable, M_i is 1-dependent upon itself.

Shortest dependency paths play an important role in establishing a bound on the size of the interval of active iteration numbers.

Definition 21: Let k be the smallest integer such that node v_j is k-dependent on v_i. Then we say v_j is *of distance k* from v_i.

Note that if v_j is of distance k from v_i all paths from v_i yo v_j contain at least k *D* operators, counting v_i, but not v_j.

Theorem 22: A loop code-block $\mathcal{L}$ without conditionals is a bounded loop if and only if the cyclic portion of $\mathcal{L}$ forms a single, strongly connected component.

Proof: ($\Rightarrow$) Suppose the cyclic portion of $\mathcal{L}$ forms a single, strongly connected component. Let (s,t) and (u,v) be an arbitrary pair of arcs. Let l be a leading variable, as per Lemma 18. Since $\mathcal{L}$ is strongly connected, M_l is dependent on t. Let l be the distance of M_l from t. By Corollary 19, u is of distance at most 1 from M_l. Node v is of distance at most 1 from u. Since M_l is 1-dependent on itself, v is k-dependent on t, using edge (u,v), for all $k \geq l + 2$.

Let m be the number of loop variables. Then $l \leq m$. Thus, if token t_i exists for arc (s,t) and token t_j exists for arc (u,v), the iteration number of t_j can be at most $m+1$ larger than that for t_i. Since the two edges are arbitrary, no two tokens may coexist at any time with iteration numbers differing by more than $m+1$.

($\Leftarrow$) Suppose, on the other hand, the cyclic portion of $\mathcal{L}$ does not form a single, strongly connected component. Let l be a leading loop variable, *i.e.*, there exists a path from M_l to v, for any node v. Since, the $\mathcal{L}$ does not form a single, strongly connected component, there exists a node v such that M_l does not depend on v. Thus, l may perform an arbitrary number of iterations before without v firing. Since there is a path from M_l to v, an arbitrary number of tokens can accrue along this path. □

This result provides a precise, easily computable, characterization of the class of bounded resource loops. Determining whether a graph is strongly connected can be performed in O(e) operations, *cf.* [1], page 189. Such loops have predictable storage requirements, generate a bounded number of subordinate invocations, and allow iterations identifiers to be safely assigned as $i+1$ MOD $m+1$, where m is the number of loop variables.

Corollary 23: If a loop with m loop variables is resource bounded, it can unfold into at most $m+1$ concurrent iterations.

With additional effort we can compute tighter bounds on the number of concurrent iterations. The value of l in the proof of Theorem 22 can be computed using an all-pairs shortest path algorithm. Assign a weight of 0 to all edges, except the output arcs of D operators; these should be assigned a weight of 1. Compute the all-pairs shortest (i.e., least weighted) path [1]. For each node v, let l_v be the weight of the shortest path from v to a leading merge. Take l to be the maximum of the l_v. A variety of algorithms exist for all-pairs shortest path; they typically require $O(n^3)$ operations, where n is the number of nodes.

We can derive nearly as tight a bound with less computational effort by abstracting the internal structure of the loop body. Every node is on a path between two *merges*, containing a single *D* operator[18]. Consider an arbitrary node v. There exists a *merge* node M of distance at most 1 from v. Thus, the distance from v to a leading merge is at most 1 greater than the distance from M to a leading merge. We can take *l* to be the maximum distance from a merge to a leading merge plus 1. This observation is captured in the following algorithm.

Algorithm 24: Maximum Loop Unfolding

Input. Loop Code-Block Graph (cyclic portion), $\mathcal{L}$

Output. Upper bound on the number of concurrent iterations.

Method.

1. Construct the *merge* 1-dependence graph $\mathcal{M}^1$ from $\mathcal{L}$. $\mathcal{M}^1$ contains a node for each *merge* node in $\mathcal{L}$ Edge (i,j) $\in$ T if there is a path from M_i to M_j containing a single *D* operator.
2. If $\mathcal{M}^1$ does not form a single connect component, stop; the loop is unbounded.
3. Construct the edge-weighted *merge* dependency graph $\mathcal{M}^*$ from $\mathcal{M}^1$. $\mathcal{M}^*$ contains a node for each *merge*. Edge (i,j) $\in \mathcal{M}^*$ with a weight of k if the shortest path from i to j in $\mathcal{M}^1$ has length k.
4. Return the maximum weight from a *merge* node to a leading *merge*, plus 1.

To demonstrate that Step 2 is correct, we must verify that $\mathcal{M}^1$ forms a connected component if and only if the cyclic portion of $\mathcal{L}$ forms a connected component. If the cyclic portion of $\mathcal{L}$ forms a connected component, then $\mathcal{M}^1$ certainly does. Conversely, consider any pair of nodes u and v. There is a path from u to a merge and a path from a merge to v, since $\mathcal{L}$ is well-connected. Since all merges are connected, u and v are connected. Let *m* be the number of loop variables and *e* the number of edges. Step 1 requires $O(me)$ operations. Step 2 is $O(m)$. Step 3 is $O(m^3)$. Generally, *m* is much smaller than the number of nodes or edges.

[18] This is analogous to the assumption for acyclic graphs that all nodes are on a path from the source to the sink. Dataflow operators have no side-effects, so a node with no output arcs would be useless. Switches for a conditional are an exception, since certain input variables may not be required for one or the other setting. Additional arcs can be introduced to preserve the path property. In practice, impure operators are introduced which have limited side-effects; output arcs must be added to preserve the path property.

7.4. Controlling Loop Unfolding

Automatic unfolding of loops is an essential source of parallelism in the U-interpreter model. Unfortunately, loops which offer vast amounts of parallelism present the possibility of run-away parallelism, requiring vast amounts of machine resources. It is essential that these loops be able to unfold, but in a controlled fashion. The structural characterization of resource bounded loops suggests how unbounded loops can be controlled: add auxiliary arcs so that the cyclic portion of a loop code-block forms a single, strongly connected component. Unfortunately, the loops generated by this transformation offer little parallelism. The potential parallelism can be increased with the addition of dummy loop variables, but this results in substantial overhead. To provide efficient, controlled unfolding of loops, it is necessary to step somewhat outside the basic model. This section presents a series of loops transformations, aimed at efficient, controlled unfolding. The first transformation generates bounded loops from unbounded loops, with the addition of a minimum number of arcs. The second provides enhanced unfolding with the addition of loop variables. The third provides similar benefit, but with the addition of special control operators.

Transforming unbounded loops into bounded loops

Let $\mathcal{L}$ be an unbounded loop. By introducing auxiliary arcs, the graph for $\mathcal{L}$ can be transformed into a strongly connected graph $\mathcal{L}'$. The auxiliary arcs represent artificial dependencies; the data which travels along these arcs is discarded by the destination operation. Care must be exercised that the transformed graph is legal; the loop body must remain acyclic. $\mathcal{L}'$ will compute the same function as $\mathcal{L}$, but will exhibit more controlled behavior.

How can a minimum number of arcs be introduced to transform $\mathcal{L}$ into a strongly connected graph? Suppose the strongly connected components are collapsed into individual nodes. The resulting graph is acyclic. By Lemma 18, there is exactly one node which has no predecessors; this represents the component containing the leading variables. Call it the *leading node*. There are a collection of *trailing nodes*, which have no successors. Every node is on a path from the leading node to a trailing node. Suppose there are n trailing nodes. To make the graph strongly connected n arcs must be introduced, such that each trailing node is connected to the leading node. An edge can be introduced between each trailing node and the leading node, or the trailing nodes may be chained together, with the final one connected to the leading node. We adopt this latter strategy in the loop transformation algorithm below.

Algorithm 25: Bounded Loop Transformation

Input: Loop Code-Block $\mathcal{L}$

Output: Bounded Loop Code-block $\mathcal{L}'$, equivalent to $\mathcal{L}$.

Method.

1. Construct the *merge* 1-dependence graph $\mathcal{M}^1$.

2. Reduce $\mathcal{M}^1$ to $\mathcal{T}$, by coalescing strongly connected components.

3. Let L be a leading node, and $T_1, ..., T_n$ be the trailing nodes in $\mathcal{T}$. For each T_i, let t_i be a loop variable represented by node T_i in $\mathcal{T}$. Let S_i and D_i be the *switch* and *D* operators for t_i. Let l be a loop variable represented by L, with *D* operator D_l. To generate $\mathcal{L}'$ from $\mathcal{L}$:

   a. Add an auxiliary arc from S_i to D_{i+1}, for $i = 1$ to n-1.

   b. Add a control arc from S_n to D_l.

The distinction between auxiliary and control arcs is somewhat artificial; they both introduce a dependency between otherwise independent computations. We want to call special attention to the dependency from the trailing variable to the leading variable because it is the keystone in controlling the number of concurrent iterations.

As an example of the transformation, consider the loop body in Figure 7-4. It has three loop variables: I, SUM, and PROD. The *merge* dependency graph is shown in Figure 7-5. Under Algorithm 25, an auxiliary arc is added from the *switch* for SUM to the *D* for PROD. A control arc is added from the *switch* for PROD to the *D* for I. This results in the loop shown in Figure 7-6. Note, that the transformation introduces a cycle of dependencies including all the *merge* operators. The new merge 1-dpendence is shown in Figure 7-7.

Let us examine how this modified code-block unfolds, assuming F and G represent lengthy computations. Initially, all the switches are enabled. They fire and enable the first instances of F and G. D_{SUM} and D_{PROD} will not be enabled for some time, but D_I in enabled immediately. It fires, allowing variable I to begin its second iteration and enables a second instance of F and G. The loop does not continue to unfold, however, as the unmodified loop would. The first instance of G must complete and allow PROD to begin its second iteration, before I can begin its third. At most three instances of F and two instances of G can be in execution simultaneously.

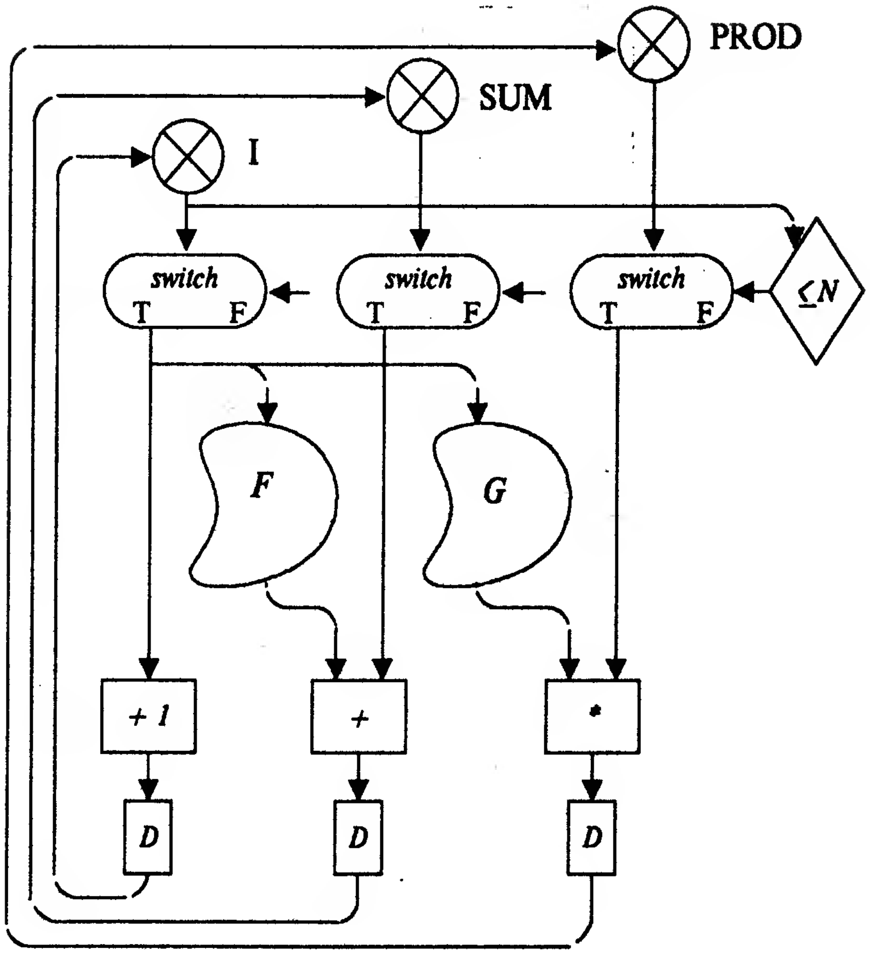

Figure 7-4: Unbounded Loop

Increased parallelism in bounded loops

The example above demonstrates essentially maximal unfolding for a bounded loop; the degree of unfolding is equal to the number of loop variables. To allow such a loop to unfold further, it is necessary to introduce auxiliary loop variables and thus weaken the coupling between the trailing variable (PROD) and the leading variable (I).

In the example above, auxiliary loop variables can be introduced as shown in Figure 7-8. Each new loop variable increases the maximum number of concurrent instances of each F and G by one. Adding a loop variable involves adding three new operators (*merge, switch, and D*) and an arc from

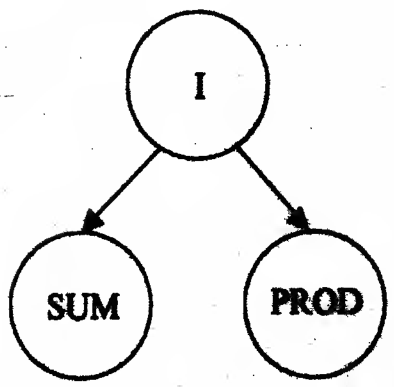

Figure 7-5: Merge-1 Dependency Graph

the switch of the trailing variable t_n to the new *D*. The control arc ties the new *switch* to the leading *D*.

This approach is effective, but has a number of shortcomings. It introduces a fair amount of overhead. The lag time from the completion of an iteration of the trailing variable to the initiation of a new iteration of the leading variable increases with the number of auxiliary variables. Finally, the degree of unfolding is fixed in advance; the approach can not used to apply dynamic control over the amount of parallel activity. A more flexible strategy which allows the degree of unfolding to be easily adjusted with little overhead is desired. However, by Corollary 23, this can not be accomplished within the basic model.

Control operators

The essential contribution of the dummy loop variables is to provide a source of trigger tokens along the path from the trailing variable to the leading variable. This can be accomplished by initializing the graph with k tokens (with iteration numbers 1 through k) along the original control arc. A special D^k operator (which increments the iteration number by k) must be introduced so that the i^{th} iteration of the trailing variable triggers the $i+k^{th}$ iteration of the leading variable. This approach solves the problems raised above, but compromises the basic model somewhat, since graphs are not self-cleaning and self-initializing.

It is important to draw a distinction at this point between the abstract model (the U-interpreter)

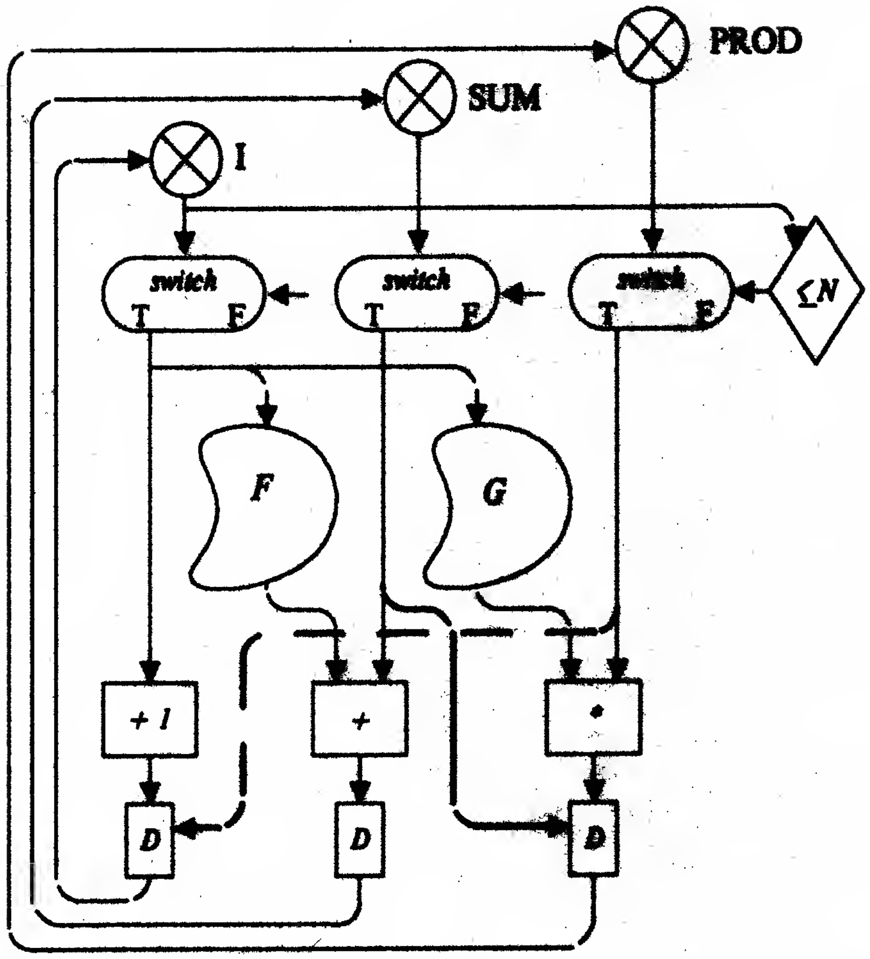

Figure 7-6: Transformed (Bounded) Loop

and a practical realization. There is no reason to constrain loops in the abstract model, since resources are unbounded. The only reason to constrain loops is to force programs to execute within the resource constraints of a concrete machine. The machine, unlike the abstract model, associates a certain amount of state information with each code-block activation. This can be used to implement the control operators required here.

A control operator is needed which acts like a governor. It should allow the leading variable to circulate until it becomes k iterations ahead of the trailing variable, at which point the leading variable must be inhibited until the trailing variable completes another iteration. Call this a *G* operator. It can be implemented with a bit-list and enough storage for a single token. The *G*

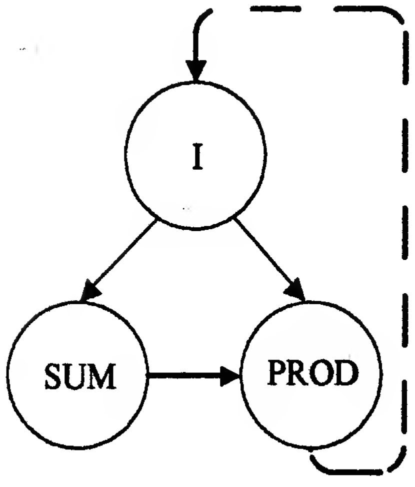

Figure 7-7: Transformed Merge-1 Dependency Graph

operator has two inputs, one for the leading variable and one for the trailing variable. It fires whenever a token arrives on either input. The bit-list is used to keep track of the trigger tokens that arrive from the trailing variable. Initially, all the bits are on. When it fires for the leading variable, the bit-list is examined. If the next iteration identifier is free, the leading variable passes through and the bit is cleared. Otherwise, the leading variable is stored internally until the next iteration identifier is free. The arrrival of a trigger token resets the corresponding bit. Certainly other implementations are possible, but the point is that providing a certain amount of storage for each code-block invocation is much simpler than purging trigger tokens from the waiting-matching stores. The approach suggested here yields the graph in Figure 7-9 for the example presented above.

7.5. Storage Requirements of Parameterized Loops

The transformation above generates loops with parameterized unfolding. The maximum number of concurrent iterations is a parameter *k*, which can be set when a loop code-block is invoked. The token storage requirements of such a loop depend on the value of *k*. For any particular *k* the storage requirement can be determined by solving Linear Program 12. However, we should like a more efficient way of computing the storage requirement as a function of *k*. This can be accomplished with a slight extension of the linear programming technique developed above, called

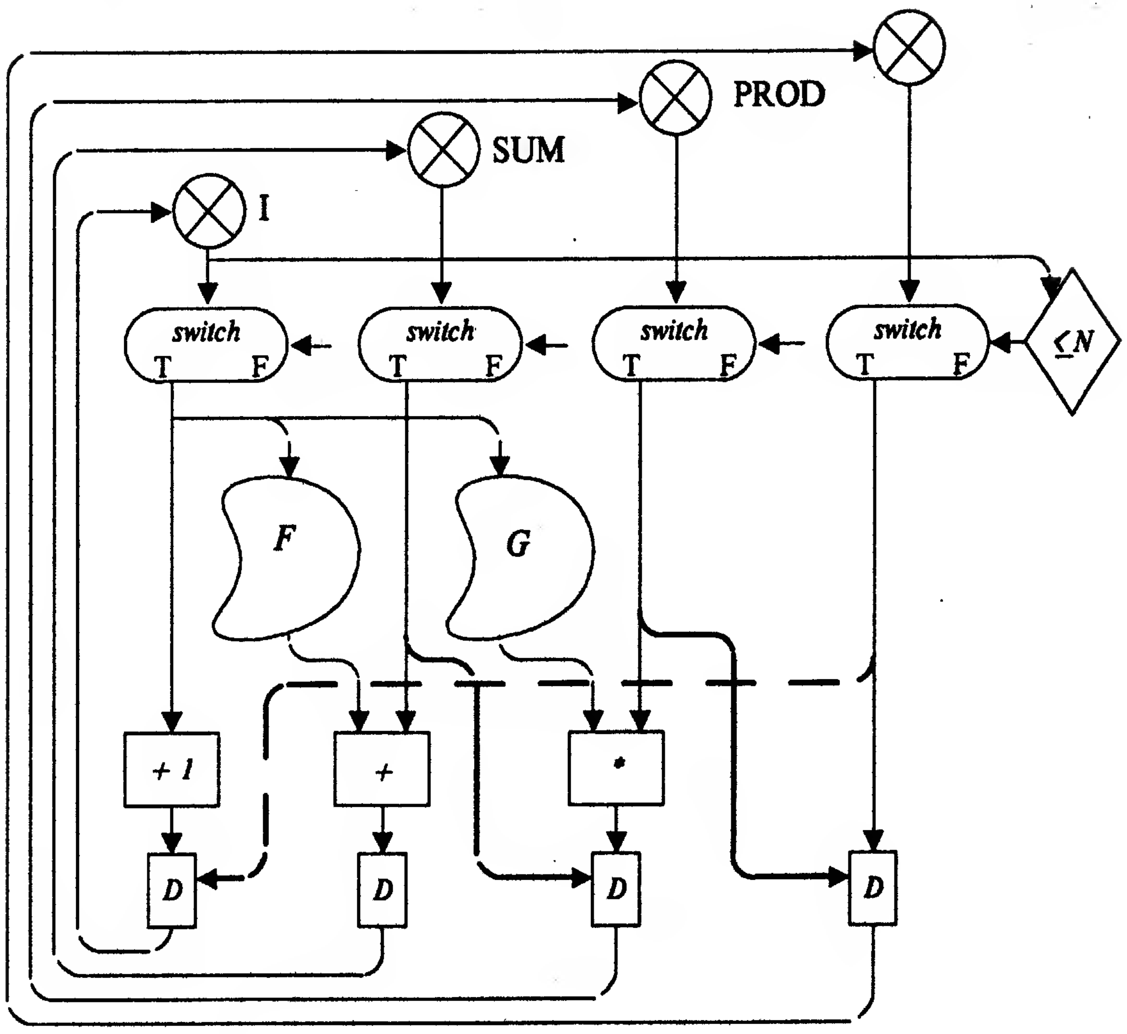

Figure 7-8: Auxiliary Loop Variables

parametric programming [13]. The linear program is first solved for $k = 1$; the solution is extended incrementally for larger values of k.

Intuitively, for sufficiently large k, the storage requirement should increase linearly with k. Recall the loop structure that gives rise to unbounded storage requirements. Certain loop variables circulate freely while others, which depend on the circulating variables, fail to circulate. Each iteration of the circulating variables deposits a token on the arcs which represent the dependence to the non-circulating variables. We expect the worst case token-storage requirements to be of the

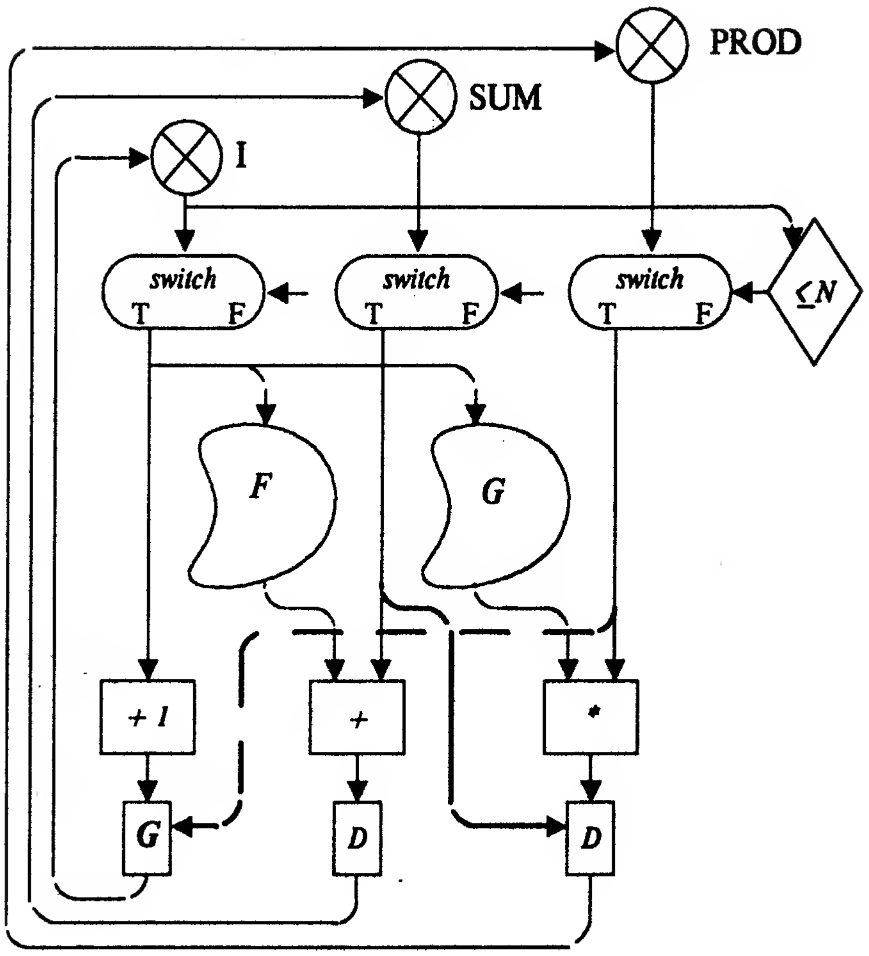

Figure 7-9: G-Transformed (Bounded) Loop

form $ak + b$ where k is the number of iterations, for sufficiently large k. The constant coefficient b accounts for a constant number of tokens in the components of the circulating variables. The linear coefficient a is the number of arcs upon which tokens accumulate. For small values of k the storage requirement behaves more erratically, because k can influence which components circulate and which lag behind. This intuitive viewpoint is useful, but we should like a more concrete result. The basic algorithm is given below; the interested reader is referred to [13], Chapter 3.

We are given a linear program for a parameterized loop code-block. The parameter k appears on the right-hand side of the constraint corresponding to the control arc, say row m.

1. Transform the linear program into *canonical form* by adding slack variables and converting inequality constraints to equality constraints. (The constraint matrix is still totally unimodular.)

2. Solve the linear program for k = 1. The final tableau appears as in Figure 7-10. Increasing k by Δk increases the token storage requirement by $-c_{n+m}\Delta k$, for Δk small enough that the current basis remains optimal.

3. Determine the range of Δk for which the current basis remains optimal. This is given by:

$$\mathrm{MAX}_i\{\,-r_i/b_{im} \mid b_{im} > 0\,\} \le \Delta k \le \mathrm{MIN}_i\{\,-r_i/b_{im} \mid b_{im} < 0\,\}, \text{ for } i = 1 \text{ to } m.$$

4. Increase *k* to the point where a pivot must be performed. The binding constraint, say row r, determines which variable will leave the basis. Apply the usual ratio test to determine which variable will enter the basis. Perform the pivot.

5. Repeat steps 3 and 4 until the maximum permissible value of *k* is reached or until no upper limit is placed in step 3.

$$
\begin{array}{l}
c_1x_1 + c_2x_2 + \cdots + c_nx_n + c_{n+1}x_{n+1} + \qquad + \cdots + c_{n+m}x_{n+m} = z \\[1ex]
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n + b_{11}x_{n+1} + b_{12}x_{n+2} + \cdots + b_{1m}x_{n+m} = r_1, \\
a_{11}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n + b_{21}x_{n+1} + b_{22}x_{n+2} + \cdots + b_{2m}x_{n+m} = r_2, \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n + b_{m1}x_{n+1} + b_{m2}x_{n+2} + \cdots + b_{mm}x_{n+m} = r_m.
\end{array}
$$

Figure 7-10: Tableau After Solving for a Particular k

Heretofore, the focus has been on determining the resource requirements of a loop. This parametric programming technique allows the converse question to be addressed as well. Given that a loop must execute within a prescribed amount of resources, to what extent can it be allowed to unfold. The parametric programming technique determines the bound. The loop transformations enforce the bound.

7.6. Extensions for Fairness Assumptions

It is important at this point to evaluate the assumptions under which worst-case token storage requirements are determined in the approach presented above. The constraint systems presented above define the space of all possible legal configurations, and the storage requirements have been determined over this space. This allows for arbitrary delays from the time an activity is enabled until the time it actually fires. This approach was adopted because communication delays are extremely unpredictable in an asynchronous system such as the Tagged Token Dataflow Architecture. However, in some ways it is overly pessimistic. The execution sequence carried out by the Tagged Token Dataflow Architecture will tend to be much more fair. Activities get enabled and carried out in roughly a FIFO order. This restricts the space of legal configurations significantly.

In some cases, the lag between certain loop variables is bounded by virtue of the FIFO constraints, even though it is theoretically possible for arbitrary lags to develop. The summation example in Chapter 2 is a case in point. Suppose the function F is simple enough that the time required to compute it can be bounded in advance at k times the time required to increment I. Then at most k concurrent activations of F can be in execution simultaneously. Thus, assuming sufficient computational resources are available to support k concurrent activations of F, the trailing variable SUM can only lag the leading variable I by k iterations. The constraint system can be augmented to reflect this fact by adding the constraint:

$$f(M_{sum}) - f(M_I) \leq k.$$

The parametric programming technique can be employed to determine the sensitivity to k. Note that the resulting constraint system is identical to that which results from adding a control arc with k trigger tokens.

Employing FIFO constraints, rather than explicit constraints as suggested in Section 7.4 is rather precarious. The computation performed by the trailing variables would have to be almost trivial before such FIFO guarantees could be made. Communication between PEs is extremely unpredictable, as is the time to process structure memory requests. Moreover, the amount of parallel activity in the machine influences the relative speeds of various computations. The gains in efficiency over the transformed loops that result from the algorithm in Section 7.4 are offset by the hazards that may arise from inaccurate timing predictions.

The other kind of fairness assumption we might consider is a limit on the relative rates of loop variables. Suppose the trailing variables can be guaranteed to circulate at $1/k^{th}$ the rate of the leading variables. This scales the accumulation of tokens on the arcs emanating from the leading component by approximately (k-1)/k from the worst-case. Unfortunately, this kind of constraint can not be added to the constraint system with sacrificing total unimodularity.

7.7. Extensions for Nested Loops

The techniques for analyzing loops can be extended to handle nested loops, much as the techniques for acyclic code-block are extended to handle non-recursive procedures. If the inner loops are resource bounded, the cost of the edge corresponding to the activation of the inner loop can be scaled by the storage requirements of the inner loop. The parametric programming technique can be employed for the outer loop, as discussed above, by analyzing the sensitivity to one right hand side value.

If the outer loop is a bounded resource loop and the inner loop is parameterized, the composite structure can be analyzed by a similar parametric technique. Rather than examine the sensitivity to a right hand side value, the sensitivity to the cost coefficient representing the storage requirement of the inner loop is examined.

If both loops are parameterized, the problem is somewhat more complex. The two parameters (a right hand side value for the outer loop, and a cost coefficient for the inner loop) must be adjusted simultaneously. This provides a method of determining how the pair of loops can unfold, given a bound on the resources they can use collectively.

7.8. Loops with Conditionals

We turn now to the problems raised by conditional expressions appearing within the cyclic portion of loops. Recall, with acyclic graphs the introduction of conditionals made computing tight bounds on token storage NP-complete. The same holds for loops. The branch-and-bound technique introduced in Chapter 6 will provide a reasonably tight bound for loops encountered in practice. Conditionals also complicate determining the potential unfolding of loops. As a simple example, consider the graph in Figure 7-11. The 'true' setting of the conditional allows for unbounded unfolding, whereas the 'false' setting gives a bounded loop. The dependencies between loop variables can be affected by the setting of the conditional. We must assure that loops have bounded unfolding under any setting of the internal conditionals, in order to avoid token storage overflow and run-away parallelism. Moreover, we must guarantee that only a fixed sized interval of iteration numbers are required at any time. These problems are addressed in turn.

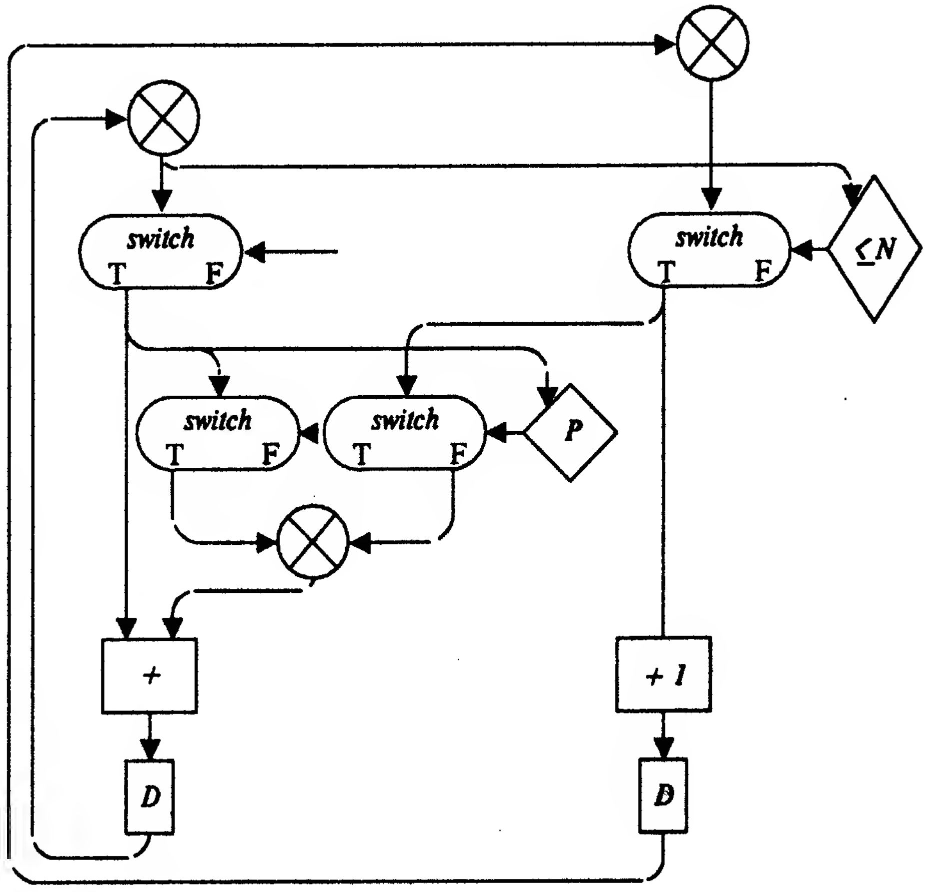

Figure 7-11: Loop with Conditional Dependencies

To assure bounded unfolding we need only guarantee that the *merge* nodes are mutually dependent (*i.e.*, the *merge* 1-dependency graph forms a connected component) regardless of the settings of the conditionals. To this end, we adopt a conservative attitude toward the dependencies implied by a conditional expression; dependencies which are affected by the settings of the conditionals are ignored. This may cause redundant arcs to be introduced when the trailing nodes are tied together, but no serious problems.

The first step is conditional extraction. Let C be a conditional expression with n *switches*, m *merges*, and no internally nested conditionals. There is an *unconditional* dependency between *switch* s_i and *merge* m_j if there is a path from s_i to m_j on both sides of the conditional. For the purpose of generating the *merge* 1-dependence graph, the conditional expression can be replaced by $n+m$ well-behaved operators, interconnected according to the unconditional dependencies. Conditional expressions are extracted, starting from the inner-most conditionals and working outward, until no conditionals remain. Apply Algorithm 25 to generate auxiliary and control arcs. The resulting loop is strongly connected, even if the conditional dependencies are ignored.

Ignoring conditional dependencies introduces a potential problem in controlling the unfolding of loops. Suppose there is a conditional dependency from the trailing variable to the leading variable, as chosen in Algorithm 25. As long as this dependence is in force, the control arc will be redundant. Increasing the lag on the control arc will not increase the potential unfolding. For this reason, it is important to keep track of potential dependencies. Auxiliary arcs should be aligned with conditional dependencies wherever possible. The control arc should not be aligned with a conditional dependency, if possible.

The other key point in Theorem 22 is the role of the leading *merge*, a *merge* node upon which all nodes are 1-dependent. Without conditionals, any *merge* which provides input to the predicate expression suffices as a leading *merge*. There is a potential problem if the predicate expression contains conditionals; perhaps one variable triggers the predicate for certain iterations and a different loop variable for others. However, this problem is resolved by requiring that conditionals be well-connected. The predicate expression yields a single result, a boolean. Thus, there is a path from any input to the node which generates the control tokens for the bank of switches. Whether or not the loop contains conditionals, any *merge* providing input to the predicate expression suffices as a leading *merge*.

7.9. Summary

Algorithm 25 meets the task set forth at the beginning of the chapter: given a U-interpreter loop, transform it into an equivalent loop that has (1) bounded, predictable token storage requirements, (2) recycles a fixed collection of iteration identifiers automatically, (3) allows only a bounded number of iterations to be active concurrently. Such a loop is particularly well-suited for execution on the Tagged Token Dataflow Architecture.

The essence of this transformation is the observation that loops have bounded unfolding if the cyclic portion forms a strongly connected component. Auxiliary arcs can be added so that this condition is met. The resulting loop is amenable to analysis using the constraint system technique developed in Chapter 5. Furthermore, the loops can be controlled dynamically.

We now have included the entire collection of graph schemata allowed by the U-interpreter. We can once again evaluate our situation with respect to the resource management problems.

1. *Termination Detection*: Solved. For any code-block, cyclic or acyclic, by adding arcs we can guarantee that the firing of the *end* node signifies termination.

2. *Token Storage Overflow*: Solved, with the caveats mentioned in the previous chapter concerning conditionals. By transforming loops in the manner described above we guarantee that every code-block invocation has bounded token storage requirements. A reasonably tight bounded on the worst-case token storage requirement of a given code-block can be determined by solving a linear program.

3. *Iteration Overflow*: Solved. Transformed loops require a bounded number of iteration identifiers and recycle iteration identifiers automatically. The *D* operator simply increments i, modulo some parameter k.

4. *Program Deadlock*: Partially solved. In restricted cases, the resource requirements of entire subtrees of the invocation tree can be predicted. In general, the requirements of the entire program can not be predicted. However, by restricting the breadth of the active invocation tree, the potential for program deadlock can be reduced. Restricting the unfolding of loops is an essential contribution in this light. Loops which exhibit potentially unbounded unfolding tend to cause the active invocation tree to grow very broad, very rapidly.

Chapter Eight

Dynamic Control

The preceding chapters provide a complete solution to the first three resource management problems: termination detection, token storage overflow, and iteration overflow. However, program deadlock has been only partially addressed. Let us review the steps taken so far. Chapters 2 through 4 established the nature of the problem. A program unfolds as a tree of code-block invocations. Each invocation requires certain resources. Program deadlock arises when the active invocation tree exceeds the resource capacity of the machine. Chapter 5 provided a means of predicting the overall resource requirements for a restricted class of programs. Chapter 6 demonstrated that overall resource requirements can not be predicted for programs in general. Thus, it is not possible to determine *a priori* whether a given program will deadlock. In light of this, we have aimed at reducing the potential for program deadlock, rather than trying to avoid it all together. The key observation is that the resource requirements of a program are reduced if the breadth of the active portion of the invocation tree is restricted. Thus, controlling the number of concurrent iterations of loops is a vital step in reducing the potential for program deadlock. If a loop can potentially unfold into a large number of concurrent iterations under reasonably fair scheduling, each iteration must involve a substantial amount of computation. Rapidly spawning many such blocks of computation makes program deadlock very likely. It would be wiser to generate only as much parallel computation as required to saturate the machine. In this chapter, we consider even more aggressive measures to reduce the potential for program deadlock and examine ways to dynamically control the amount of parallel activity.

The central concept in this chapter is the invocation tree. At any time, the active portion of the invocation tree represents the load on the machine resources. The depth of the active portion is dictated by the computation being performed. However, the breadth of the active portion is determined by the way a program is allowed to unfold. Thus, the overall size of the active invocation tree can be partially controlled. An effective resource management system should dynamically control program unfolding so that the size of the active invocation tree does not exceed the available resources. Clearly, this is not possible for all programs, since some programs simply require more resources than the machine provides. The goal is to allow a large class of programs to execute efficiently on the Tagged Token Dataflow Architecture. This requires constraining a program when it begins to generate parallel activity far in excess of the amount of parallelism that can be exploited by the underlying machine.

8.1. Breadth-first and Depth-first Evaluation

To understand how program unfolding can be controlled, let us examine two extreme scenarios. Consider first the maximally parallel scenario offered by the U-interpreter. All processor resources are assumed to be unbounded; an activity executes as soon as it is enabled; and the invocation tree unfolds in a breadth-first manner. As soon as an activity completes, all of the activities that it enables execute. At the code-block invocation level, as many branches as possible are pursued in parallel. Independent, subordinate invocations are initiated in parallel. Each subordinate invocation is the root of a subgraph of independent computation, so new subgraphs are initiated at the soonest possible moment. Each of these enable as many subordinate invocations as possible, and so on. The active portion of the execution graph tends to become broad and bushy. Assuming unbounded resources, the program executes in the minimum possible time.

At the other extreme, consider the scenario for a conventional (sequential) machine. Only a single code-block invocation can be active at any time. When a subordinate invocation is initiated, the caller is suspended until the subordinate completes. The invocation tree unfolds in a depth-first manner. A single branch of the invocation tree unfolds until it terminates. It then retracts to the last unevaluated alternative branch, and so on. It is possible for a dataflow machine to pursue a depth-first unfolding as well. Each code-block invocation is permitted to have at most one active subordinate invocation at any time. The active invocation subtree is restricted to a single branch, however, unlike a conventional machine, there may be activity all along the branch. A code-block invocation need not suspend when it invokes a subordinate.

In the maximally parallel scenario, the active portion of the invocation tree may include the entire invocation tree. In the sequential scenario, the active portion never includes more than a single branch. Thus, a maximally parallel evaluation can require exponentially more resources than a sequential evaluation.

Tagged Token Dataflow Architecture leans toward a maximally parallel evaluation, like the U-interpreter, even though it can exploit only a certain amount of parallelism. Activities are enabled in essentially FIFO order. If invocations are initiated as soon as they are enabled, independent subordinate invocations will execute in parallel, regardless of status of the machine or the amount of exposed parallelism. Consider a simple program which employs binary recursion. At the top level, both subordinates are activated; one perhaps slightly ahead of the other. They may be placed on separate processing elements or their execution may be interlaced in the pipeline of a single processing element; in either case, they make comparable progress and their four subordinates are activated nearly in parallel. The unfolding proceeds in a breadth-first manner. When the amount of exposed parallelism exceeds the amount of parallelism that can be exploited, the various invocations timeshare the processing elements at the instruction level. The progress of each of the active code-blocks is comparably degraded, and they continue to make comparable progress. The invocation tree continues to unfold in a breadth-first manner, even though the processing power of the machine is fully utilized and additional parallelism can not be exploited. If the active portion exceeds the machine resources, the program deadlocks. A less eager strategy might allow the program to execute to completion and still fully utilize the processing power of the machine.

To effectively utilize the machine and allow large programs to run on it, the active invocation tree should be just broad enough to provide enough parallelism to saturate the machine. A breadth-first evaluation can be pursued until enough parallelism is generated to saturate the machine. At that point, the various independent subtrees can be constrained to a depth-first evaluation.

How should such a control strategy be implemented? We have two choices: (i) transform graphs by adding auxiliary arcs to limit the number of requests for parallel invocations, or (ii) allow the resource management system to delay initiating invocations. Both approaches effectively limit the branching of the active invocation subtree.

Depth-first Graphs

Given an acyclic graph, either an acyclic code-block or the body of a loop, it can be transformed into a purely depth-first graph (*i.e.*, one that will have no more than one active child at any time) with the addition of auxiliary dependencies. The resulting graph must be acyclic as well. The transformation is given below.

Algorithm 26: Attenuation of Active Invocation Tree.

Input. Acyclic program graph $\mathcal{L}$, without conditionals.

Output. Equivalent Depth-first Graph.

Method.

1. Compute the transitive closure of the graph. Discard all but the *apply* nodes. This gives the dependencies between subordinate invocations.

2. Compute a topological ordering of this invocation dependency graph. If there is no dependency from an invocation a_i to its successor a_{i+1}, introduce an auxiliary arc from the *Apply*$^{-1}$ for a_i to the *Apply* for a_{i+1}.

Conditionals require additional care. First, there may be conditional dependencies between *apply* nodes. Second, auxiliary arcs can not be attached to nodes within a conditional without observing the *switch* and *merge* structure outlined in Chapter 2. We must guarantee that under any setting of the conditionals, there is a path in the graph which contains every *apply* operator that fires. The key is to tie together the *apply* operators within each side of a conditional, and then to interface these *apply* chains to the chain of *apply* operators in the enclosing graph. Let $\mathcal{C}$ be an inner-most conditional nested within graph $\mathcal{G}$. Note $\mathcal{G}$ may be an acyclic block, a loop body, or a conditional expression. Apply Algorithm 26 to the two sub-blocks of $\mathcal{C}$ to tie the *apply* nodes into linear chains. Choose a *switch* node and connect its 'true' and 'false' outputs to the first *apply* in the respective chains. Generate a new *merge* node and connect the last *apply* nodes in the two chains to it. Thus, for either setting of the conditional, the *apply* nodes in the conditional are on a path from the chosen switch to the new merge. Partition $\mathcal{G}$ into $\mathcal{G}_t$ and $\mathcal{G}_b$ such that (i) any node which is dependent (even conditionally) on a *merge* of $\mathcal{C}$ is in $\mathcal{G}_t$, and (ii) any node upon which a *switch* of $\mathcal{C}$ depends appears in $\mathcal{G}_b$. Apply Algorithm to $\mathcal{G}_t$ and $\mathcal{G}_b$ independently. Connect the last *apply* in $\mathcal{G}_t$ to the chosen switch in $\mathcal{C}$. Connect the auxiliary *merge* in $\mathcal{C}$ to the first *apply* in $\mathcal{G}_b$.

Thus, for every code-block we have two graphs, a breadth-first graph and a depth-first graph. It is not difficult to engineer the program representation so that these are represented by a single code-block; the auxiliary arcs must be specially designated. Tokens for a particular invocation carry a flag which indicates the form of evaluation. When a code-block is invoked, the resource manager can designate the form of evaluation to pursue. A breadth-first strategy can be followed until ample parallelism is generated. At that point, a more conservative depth-first strategy can be adopted, until the exposed parallelism falls below a certain threshold.

Delayed invocation requests

Delaying invocation requests within the resource manager boils down to constructing an appropriate set of requests queues. The primary issue is how to record the structure of the executing program so that given an invocation request, the part of the active invocation tree that it stems from can be determined. A simple approach is to assign a threshhold of subordinate invocations to each invocation and delay invocations when the threshold is exceeded. A queue of pending subordinate invocations is maintained as part of the state information associated with each invocation. When an invocation is requested, the number of current subordinates is examined. If the threshold is exceeded, the request is queued; otherwise, the invocation is performed. When subordinate invocations terminate, pending invocations are processed.

8.2. A Control Strategy

Given a mechanism for controlling the unfolding of program, we must develop a strategy for applying control and determine the class of program which will execute successfully on the machine under such a strategy. One obvious strategy is to limit the unfolding to a fixed number of sequential threads. The limit being determined by the amount of parallelism the machine is capable of exploiting. In the Tagged Token Dataflow Architecture, each PE is capable of exploiting approximately eight-fold parallelism. So in a configuration with p processing elements, it would be reasonable to limit the unfolding the $8p$ to $16p$ sequential threads.

We can categorize the class of program which are sure to execute successfully on the machine with such a strategy. Suppose k branches of the invocation tree are allowed to execute in parallel. What are the resource requirements of a program executing under such a strategy? In the worst-case, k independent branches are established in the top few levels of the invocation tree, and each extends the full depth of the tree. Such a program requires k times as much computational resources as a sequential evaluation. Thus, a program can execute $k \cdot p$-fold parallelism on p processing elements, if a sequential evaluation of the program can execute with $1/k^{\text{th}}$ the resources in a single processor. Thus, the space complexity of a program under pure sequential evaluation provides a good metric for whether a program will run effectively within the resources provided by a parallel machine, under such a control strategy.

Note that Processing Elements in the Tagged Token Dataflow Architecture do not pursue a pure sequential evaluation, in the sense meant here. They tend to pursue a breadth-first evaluation, as explained above. The fewer the number of processing elements, the more restricted the breadth of the active invocation tree should be.

The relationship between resource requirements and program unfolding outlined above has important consequences. Consider the implications for the viability of large systems of small processors. Such systems can execute only a small class of programs effectively, namely programs whose invocation tree is extremely broad and shallow. Adding processors to such a system allows for a broader active invocation tree, but not a deeper one. To support a deeper tree, either the processors must be larger, or less parallelism exploited. If parallelism is restricted too much, processors will idle; their resources being used simply to record the state of the computation.

In summary, the amount of exposed parallelism should be constrained to be some multiple of the number of processors. Since the resources required to execute a program can grow linearly with the amount of parallelism exposed, such a policy maintains a close match between the supply and demand for resources. A simple rule of thumb applies: if a program can execute successfully in a sequential evaluation on a single processor, it should execute successfully with a limited parallel evaluation on any number of processors, each providing resources equivalent to the singleton processor.

Note that at the code-block invocation level, depth-first evaluation requires the least amount of resources. At the activity level, this is not the case. Consider a loop code-block such as the summation example from Chapter 2. With a depth-first evaluation, the leading variable (I) would perform all n iteration before any other operators had a chance to fire. Even if the function F required very little time to compute, the resource requirement would be a multiple of n. Note, that with bounded resource code-blocks, the choice of scheduling strategy at the activity level has less dramatic effects.

Finally, note that less resources are required if branching is permitted toward the bottom of the tree, rather than toward the top. This is a bit difficult to arrange, however. The invocation tree is not known in advance, so it is impossible to determine when the bottom is near. Also, it introduces a time/space compromise. In order to allow for branching near the bottom of the tree, a deep sequential thread must be allowed to develop.

8.3. A Resource Management Policy

The results of this chapter, and preceding ones, can be combined in an overall resource management policy. As part of the compilation phase, acyclic code-blocks are analyzed to determine their token-storage requirements. Cyclic code-blocks are transformed to introduce control arcs and analyzed to determine their storage requirements, as a function of the control parameter. Code-blocks are augmented with specially designated arcs to allow for breadth-first or depth-first evaluation.

Each processing element tries to maintain k independent branches of the invocation tree. Note that an invocation in depth-first mode extends a branch, whereas in breadth-first mode an invocation introduces a new branch. A PE which falls below its threshold operates in a breadth-first mode to increase its load. It may also signal other PEs, to indicate that it needs work. A PE which exceeds its threshold operates in depth-first mode. Note that this does not alleviate the excess, it simply inhibits further increase. The work can decrease in two ways: a sequential thread can die out, or one can be given away to another PE.

Chapter Nine

Conclusion

The U-interpreter provides a powerful and elegant formal model for dataflow computation. It offers a precise concept of legal program execution, abstracting away all details concerning the management of computational resources. The U-interpreter is almost too successful in abstracting the role of resources in dataflow computation; even though the model has been well represented in the literature for some years, no clear notion of the resource requirements of dataflow programs has emerged. This thesis offers such a notion. One must understand, however, that the resource requirements of a dataflow program depend heavily on the execution order, and hence on the amount of parallelism that is exploited. Realizing the U-interpreter in a concrete machine involves tackling certain basic resource management problems that are subterfuged in the formal model, including: distribution of work, termination detection, token storage management, tag management, and control over program unfolding. These issues are addressed in this thesis in the context of the Tagged Token Dataflow Architecture.

Understanding the Abstract Model

Although the motivation for the thesis is effective use of the Tagged Token Dataflow Architecture, the contributions it offers toward understanding the abstract model are important as well. The firing rule and the graph schemata are well represented in the literature [9, 3]. However, the focus on the invocation tree as means of understanding the dynamic behavior of dataflow programs has not been discussed in the literature. It is an important concept for a number of reasons: (i) it makes clear the role of activity names, (ii) it suggests how activity names can be represented by smaller tags, (iii) it provides a framework for describing the resource requirements of programs, and (iv) it suggests how the resource requirements of programs can be controlled. It has been argued in the literature [12] that dataflow models are extremely amenable to formal analysis. This viewpoint is born out extensively in the thesis; powerful algebraic and graph theoretic techniques have been employed to analyze the nature of dataflow programs. The applications for these techniques extend beyond the work presented here.

Realization of the Model

In realizing the U-interpreter, there is a clear appeal to maintaining the qualities of the abstract model as much as possible. This is certainly a motivating force in the design of the Tagged Token Dataflow Architecture. However, extreme caution is required, since the model can not be realized completely. The differences between the model and the machine must be carefully understood and addressed. The Tagged Token Dataflow Architecture captures the firing rule precisely, and storage for token is allocated and released automatically by the hardware. Unfortunately, the finite size of the token store represents a serious hazard. Iteration identifiers are treated much as in the U-interpreter, except they are finite and rather small in size. This also presents a serious hazard. Programs are allowed to unfold automatically, as in the U-interpreter. Since the resource requirements of a program increase as more parallelism is exposed, allowing invocations to be performed at the earliest moment is a hazardous strategy. The main thrust of this thesis is overcoming these problems.

The approach we have adopted is to work within the framework of the abstract model as much as possible, because it provides a clean framework for formal analysis. Problematic programs can be transformed into equivalent programs with more tractable requirements. Termination detection is accomplished by embellishing graphs with auxiliary dependencies. A constraint system technique was developed which allows the token storage requirements of individual code-block invocations to be determined efficiently. Recognizing that a certain class of loops is particularly well-behaved led to a program transformation which allows the unfolding of loops to be controlled and iteration identifiers to be recycled automatically. Observations concerning the relationship of resource requirements to the unfolding of the invocation tree led to heuristics for controlling the unfolding of programs in accordance with the resource capacity of the machine. Thus, by analyzing and transforming program graphs, we can control the resource requirements of dataflow programs. This will allow large programs to execute effectively on the Tagged Token Dataflow Architecture.

[redacted]

Future work

This research opens doors to future work in a variety of areas. A few are enumerated below.

1. *Distribution of computation:* The techniques developed here provide a framework that should allow large programs to run effectively on the Tagged Token Dataflow Architecture. These should be integrated into the compilation process and the resource manager described in Chapter 3. Serious questions of efficiency remain. How should invocations be assigned to domains for efficient execution. A useful model of interaction between portions of dataflow programs needs to be developed, supplemented with empirical investigation.

2. *Data structures:* The results presented here should be extended to address data structures. Some aspects of the work carry over directly. For example, the core of many numerical programs is a loop which repeatedly transforms a large intermediate data structure. Each iteration creates a new structure and computes the elements based on the structures generated by previous iterations. Since loops can unfold, numerous versions of the intermediate structure are required. The techniques for controlling loops in Chapter 7 suggest how the number of intermediate structures can be limited. Also, the results of that chapter suggest how to reuse structures, with the guarantee that the previous version of a structure is no longer required at the point it it reused. The notion of resource bounded programs should be re-examined while considering data structures as well.

3. *Controlling programs:* We have developed a framework for controlling programs and some rough estimates on the resource requirements of programs under a simple control strategy. A significant amount of empirical investigation is required to flesh out these heuristics.

4. *Resource bounded programs:* We have developed the concept of resource bounded programs, but have not been extremely rigorous about it. This concept deserves to be set forth much more precisely.

5. *Alternative architectures:* We have adopted the approach of analyzing and transforming programs so that they may run effectively on the Tagged Token Dataflow Architecture. With the understanding of the resource requirements of dataflow programs developed here at our disposal, it would be valuable to re-evaluate the original design choices in the Tagged Token Dataflow Architecture. This understanding has already led to two important modifications of the architecture and a clear rationale for determining the relative amounts of certain resources. Is hardware managed token storage really the right approach? Would it be advantageous to allocate blocks of token storage explicitly to each code-block invocation? To what extent should tags reflect physical resources?

References

1. Aho, Hopcroft, and Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

2. Arvind. Decomposing a Program for Multiple Processor System. Proceedings of the 1980 International Conference on Parallel Processing, August, 1980, pp. 7-14.

3. Arvind, and Gostelow, K. P. "The U-interpreter". *COMPUTER 15*, 2 (Feburary 1982), 42-49.

4. Arvind and Brock, J.D. Streams and Managers. 217, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., June, 1982. To appear in Proceedings of the 14th IBM Computer Science Symposium..

5. Arvind, and D. E. Culler. Why Dataflow Architectures. Proceedings of The 4th Jerusalem Conference on Information Technology, May, 1984, pp. 27-32. Also appeared as MIT CSG Memo 229-1.

6. Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali. The Tagged Token Dataflow Architecture. To be published as an MIT Technical Report.

7. Arvind, and K. P. Gostelow. A Computer Capable of Exchanging Processors for Time. Information Processing 77: Proceedings of IFIP Congress 77, IFIP, August, 1977, pp. 849-853.

8. Arvind, and Gostelow, K. P. Some Relationships Between Asynchronous Interpreters of a Dataflow Language. In E. J. Neuhold, Ed., *Formal Description of Programming Languages,* North-Holland, New York, 1977.

9. Arvind, K. P. Gostelow, and W. Plouffe. An Asynchronous Programming Language and Computing Machine. 114a, Department of Information and Computer Science, University of Californiav, Irvine, California, December, 1978.

10. Arvind, and R. A. Iannucci. A Critique of Multiprocessing von Neumann Style. Proc. of the 10th International Symposium on Computer Architecture, June, 1983.

11. Arvind, and R. E. Thomas. I-Structures: An Efficient Data Type for Functional Languages. TM-178, Laboratory for Computer Science, MIT, Cambridge, Mass., September, 1980.

12. Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs". *Communications of the ACM 21*, 8 (August 1978), 613-641.

13. Bradley, Hax, and Magnanti. *Applied Mathematical Programming.* Addison Wellsley Publishing Co., 1977.

14. Coffman, E. and Denning, P.. *Operating System Principles.* Prentice Hall, Inc., 1973.

15. Dennis, J. B. *Lecture Notes in Computer Science.* Volume 19: First Version of a Data Flow Procedure Language. In *Programming Symposium: Proceedings, Colloque sur la Programmation,* B. Robinet, Ed., Springer-Verlag, 1974, pp. 362-376.

16. Garey, M., Johnson, D, and Stockmeyer, L. "Some Simplified NP-Complete Graph Problems". *Theoretical Computer Science 1* (1976), 237-267.

17. Heller, S. and Arvind. Design of a Memory Controller for the MIT Tagged Token Dataflow Machine. CSG Memo 230, MIT Laboratory for Computer Science, October, 1983. Presented at IEEE/ICCD '83.

18. Leiserson, C. and Saxe, J. "Optimizing Synchoronous Systems". *Journal of VLSI and Computer Systems 1,* 1 (1983), pp. 41-67.

19. Leiserson, C., Rose, F., and Saxe, J. Optimizing Synchoronous Circuitry by Retiming. Third Caltech Conference on VLSI, Rockville, Maryland, 1983, pp. 87-116.

20. Papadimitriou, C. and Steiglitz, K.. *Combinatorial Optimization.* Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1982.

21. Weng, K.-S. An Abstract Implementation for a Generalized Dataflow Language. TR-228, LCS, May, 1979.